UNIVERSITY OF OSLO
Department of Informatics

Improving cloud performance by solving scalability limitations in libvirt

Jon Martin Sigvaldsen

Network and System Administration

Oslo University College

May 23, 2013

Improving cloud performance by solving scalability limitations in libvirt

Jon Martin Sigvaldsen

Network and System Administration
Oslo University College

May 23, 2013

Abstract

At the heart of OpenStack, the worlds largest open cloud framework, lies libvirt. This C library is responsible for server consolidation, which again is the key to efficient utilization of the underlying hardware. However, preliminary experiments has showed that libvirt proved to scale poorly when handling large numbers of virtual machines. As libvirt is a core feature used within a multitude of applications, including several cloud services, it is essential that it scales well. Through this thesis the performance of libvirt is investigated through experiments, communication with the libvirt development team, profiling and as well as tests trying to isolate potential issues.

The issue turns out to be related to NUMA architectures, which is a hardware setup where different CPU cores have their own local memory. While this removes a potential bottleneck when it comes memory access, it introduces new complications for the software. Software used on machines with a NUMA architecture needs to be aware of the architecture in order to perform well. Upgrading software to be NUMA aware is an ongoing process with constant improvements.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor, Alfred Bratterud, for his guidance, support, motivation and encouragement. His help and expertise has been crucial throughout this project. Secondly I would like to thank Erik Blake and Daniel Berrange at Red Hat's virtualization team. They proved very helpful, and provided important insight which helped the project moving forward. Finally I would like to thank my family and friends for their continued support.

May, 2013
Jon Martin Sigvaldsen

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The requests for powerful hardware may vary greatly based on special requirements. Imagine a mathematician wanting to perform some calculation which requires a lot of computing power. One moment he is in need of very powerful hardware in order to complete the calculations, but once he is done with the calculation he no longer has a use for it. The same thing can happen with webservers. What if a major news website links to a website hosted on a single medium performance server? Suddenly the requests to the webserver will skyrocket, which may result in the webserver breaking down. Unfortunately this will happen just as the website has the potential to get a lot of views.

Such variations in hardware requirements was present at Amazon. Amazon started out as an online bookstore. In order to provide a high quality service even throughout the peak seasons, such as the Christmas time, they needed very large server parks. However, a lot of the time the large server park they had acquired was not actually needed. How can this hardware be utilized during the off-seasons, when Amazon no longer actually needs it?

Their solution was to rent it out as a cloud service. By creating a cloud on their servers they were able to rent out hardware power to everyone in need of it. This meant that they would profit from having a large server park throughout the whole year, and people which needed powerful hardware resources for a limited period was given a cheap and efficient solution.

OpenStack is the worlds most popular open source software stack for creating a clouds. It is supported by multiple large companies, including Dell, HP, Rackspace and NASA. This solution gives anyone the opportunity to set up their own cloud and utilize their hardware resources as efficiently and professionally as Amazon.

libvirt is the de facto standard C library for managing virtual machines. It is one of the underlying technologies within OpenStack, as well as several other applications and cloud computing solutions. The library is used for starting, stopping, deleting as well as migrating virtual machines. However, preliminary tests have shown that libvirt scales poorly when it comes to handling large numbers of virtual machines. Having such limitations may prove detri-

mental for libvirt in the future. Considering the large server parks that run clouds, even small improvements in performance may prove to result in large savings with regards to power consumption and hardware expenses.

Before this project started, the libvirt developers where contacted by my supervisor and asked about possible causes for the problem. They already knew that certain circumstances could cause performance bottlenecks, and proved to be interested in getting further information about the issue, as seen in appendix D. Through this project the libvirt performance issues will be investigated and ways of improving upon the issue will be looked into.

1.2 Problem Statement

The following problem statement was chosen for this project:

What causes performance issues when using libvirt to create large numbers of virtual machines? How can these performance issues be migitated?

The focus of the project is to find the cause of the performance issue, and hopefully find ways which allows a host to run more virtual machines with better performance.

1.3 Thesis Outline

This paper is divided into six chapters.

Chapter one is the introduction chapter, which describes the motivation and problem statement for the project.

Chapter two is the background chapter, aimed at giving a brief overview of the different tools used throughout the project, as well as relevant technologies.

Chapter three is the approach chapter, which describes the steps needed in order to carry out the experiments and profiling performed through this project.

Chapter four is the results chapter, presenting the results from the multiple experiments and the profiling which was performed.

Chapter five is the discussion and analysis chapter, which reflects upon the findings and suggests future work which may still remain within this area

Chapter six is the conclusion chapter stating the final conclusions of this project.

Chapter 2

Background

2.1 Tools

A number of solutions are available in order to virtualize machines. Both free and proprietary software exists, and multiple types of virtualization can be done. Desktop solutions, such as VirtualBox [1] and VMware Workstation [2] offer a way to simply install some software which lets you create virtual machines on your own computer. Other solutions do not rely on an already installed OS, but are instead installed without the need of an OS, such as Proxmox or VMware ESXi.

The focus of this thesis is on virtual machines managed through libvirt. Specifically, machines virtualized through QEMU with KVM used in order to enhance the performance of the virtual machines. This is a freely available and commonly used solution on linux systems which provides good performance on the virtual machines.

2.1.1 libvirt

libvirt is a virtualization API which provides a common and stable way to control virtual machines. Different hypervisors require different commands, which may also be changed over time. By using libvirt instead of the commands related to the specific hypervisor you get a more stable way of controlling them, and you dont have to learn a whole new set of commands if you were to switch hypervisor. Many hypervisors are supported, such as KVM/QEMU, Xen, LXC, VirtualBox and VMware ESX and GSX.

libvirt can also be used as a stable building block when creating applications. By itself it does not provide high level features, but it aims to be designed in a way that such features can be implemented on top of libvirt [3]. Virtual Machine Manager is one example of a commonly used program that is built on top of libvirt [4], but it is also an underlying technology used in a multitude of cloud solutions.

When installing libvirt through apt-get in Ubuntu 12.04, version 0.9.8 will be installed. This will be the version used for most experiments, but version 1.0.5 will also be tested in order to see if improvements have been made.

2.1.2 virsh

virsh is a command line utility which allows management of virtual machines from the command line. If virsh commands are being used, libvirt is actually being used, as virsh is provided as a part of the libvirt API.

By using virsh it is easy to collect information about the virtualized hardware of a virtual machine as an XML-file, or create new machines with the hardware specified in a XML-file. [5]

Through the command:

```
# virsh list
```

all running domains will be listed. By appending the *-all* option all libvirt managed domains will be listed, including the ones that are not currently running. This list shows command ID, name and state of the virtual machines. The ID or name can be used in order to retrieve an XML-file of a domain with the following command:

```
# virsh dumpxml <ID or name>  > <domain.xml>
```

With this XML file new virtual machines with the same specifications can easily be created.

```
# virsh create <xml.file>
```

It is also possible to make changes in the XML-file in order to customize the specifications. An advantage of doing changes through an XML-file is that every option will be available for editing.

2.1.3 QEMU

QEMU is open-source software which is capable of hardware emulation and virtualization.

As a machine emulator it will emulate some set hardware. This can allow you to see how some program run under a specified set of hardware, or allow you to run programs which are not designed for the type of hardware you are using.

Used as a virtualizer it can take advantage of the KVM kernel module in order to achieve near native performance. [6]

An example of QEMU usage could be to create an image file where you can install some OS, and then booting a emulated machine with this image as its hard drive. A separate iso file with a OS can be specified as its CD-rom. This can be done through the following commands:

```
# qemu-img create <image.file> 3G
# qemu-system-x86\_64 -m 512 -hda <image.file>
        -cdrom <operating-system.iso> -boot d
# qemu-system-x86\_64 -m 512 -hda <image.file>
```

With the first line an image file, with a capability of 3 Gb will be created. This file will not use 3 GB of your hard drive space, but can become as large as 3 GB when you install the OS and store data within it. The second line starts an emulated machine on the image created through the first line. Here we specify that the emulated machine shall have 512 MB of memory with the *-m* option, and that file.iso should be mounted as a CDROM with the *-cdrom* option. *-boot d* specifies that the CDROM should have first boot order. With this command an emulated machine will be started, and you will have to go through regular procedures in order to install the OS. At the end an OS installation you will be asked to reboot the system. Rebooting the system will not work, but when the machine is shut down you can start it up again with the third command. With an OS installed to the image file the virtual machine is now ready for use.

Many other options can be added to the commands. One we will need for our later experiments is the *-nographic* option. This option will disable graphical output and redirect the serial port to the console. By creating minimal virtual machines which then write something to the serial port we can retrieve output in order to be sure that they boot properly. [7]

2.1.4 KVM

Kernel-based Virtual Machine is a solution that allows you to use the processor of your system directly. This will greatly enhance the performance of virtual machines. A requirement for this is that the processor must support a virtualization extension, which will either be Intel VT or AMD-V based on the processor type. [8].

The virtualization extension is enabled in the BIOS of the host machine, and will by default be turned off. It should also be noted that some settings may need to be turned off in order for KVM to be supported. In the case the desktop computer used for experiments in this project, Thrusted Execution could not be enabled.

2.1.5 Perl

Perl is a well-known, widely supported high-level programming language. It was first released in 1987 and development is still progressing. It has good documentation as well as a large community availble for support. [9] The Comprehensive Perl Archive Network (CPAN) gives access to over 25000 extensions, which can simplify the tasks you want to perform. [10] Perl scripts will be used in order to set up a lot of virtual machines automatically. In order to do this, some extensions will also be used.

Sys::Virt

Sys::Virt is a module written by the libvirt developers, which lets you manage a libvirt hypervisor connection. This allows for creating scripts that uses the connection as an object, rather than specifying that the script should run specific commands. [11]

Data::UUID

The Data::UUID module is simply used for generating Universally Unique Identifiers. When creating multiple virtual machines through libvirt this is used in order to give every machine their own identifier. [12]

2.2 Profiling

In order to further examine the causes of the performance issues, profiling will be performed. Profiling involves measuring different statistics of what happens when running some code. This can mean measuring time used within different functions or finding out which routines uses most memory.

2.2.1 Devel::NYTProf

Devel::NYTProf is a perl source code profiler. By running NYTProf it is possible to see where the time was spent within a specified perl script, and determine where performance issues may be lie. The results from NYTProf can be presented through HTML-files. [13]

2.2.2 OProfile

OProfile is a profiler for Linux systems, capable of profiling all parts of a running system. It can be ran in the background in order to determine general problems on a system, or it can be used to watch specific code. OProfile is still in an alpha status, but it has been proven to be stable on a wide range of hardware setups [14]

2.3 MicroMachines

The focus of the experiments is on the virtualization software, not on the actual virtual machines. As such, it is preferable to keep the virtual machines as small and lightweight as possible. In order to achieve this, extremely simple disk images will be used. My supervisor, Alfred Bratterud, has been working on creating such images through assembly code. From his github repository, a selection of minimal images can be found. The different images can simulate different types of behavior, and they are also capable of communication over a serial connection or over a network. The machines require very little resources, and their purpose is very specific [15].

Below is the code for the image used for experiments, *serial_print.asm*:

```
boot:
mov dx, 0 ;Select COM1
mov ah, 1  ;1 -> Serial print
mov al, '!'
int 14h
```

```
run: ;Boot sequence done, do work
hlt     ;Here, just idle
jmp run ;...forever

.done:
ret

times 510-($-$$) db 0 ;
dw 0xAA55
```

At boot the image will print a single *!* to the serial port, and then it will jump onto an endless loop. This loop contains a halt command, which tells the system to wait for a clock interrupt. These interrupts happen approximately 18 times a second. Because the machines will sit in an endless loop with a halt command they will consume a very little CPU time.

By using an image with such behavior it is possible to confirm that virtual machines created actually boot successfully. By writing serial output of the virtual machine to a file, this file can then be read in order to see if the exclamation mark has been written. As mentioned earlier, the *-nographic* option will redirect the serial port to the console. When starting a virtual machine from this image with the *-nographic* option we get the following output:

```
# qemu-system-x86_64 -hda serial_print.hda -nographic
!
```

The first line boots a machine from the image and specifies that serial port should be redirected to the console. The second line appears after a short delay. This is the output from the virtual machine, which indicates that the machine has booted successfully and is now running the endless loop with the halt command.

2.4 Virtual machine customization

With the tests performed in this project some of the XML features are important to note. Below is a full XML file which can be used to create a virtual machine:

```
<domain type='kvm' id='1'>
  <name><A unique name for the virtual machine></name>
  <uuid><A UUID for the virtual machine></uuid>
  <memory>16384</memory>
  <vcpu>1</vcpu>
  <os>
    <type arch='i686' machine='pc-1.0'>hvm</type>
    <boot dev='hd'/>
  </os>
  <devices>
```

```
    <emulator>/usr/bin/qemu-system-x86_64</emulator>
    <disk type='file' device='disk'>
      <driver name='qemu' type='raw'/>
      <source file='<some disk image>'/>
      <target dev='hda' bus='ide'/>
      <address type='drive' controller='0' bus='0' unit='0'/>
    </disk>
    <serial type='file'>
      <source path='<a path for output from serial port>'/>
      <target port='1'/>
    </serial>
  </devices>
</domain>
```

There are a couple of things to note here. First off, virtual machines can be created with and without KVM. In order to function with KVM disabled, the domain type, on the first line, needs to be changed from *kvm* to *qemu*.

Another change can be done in order to limit the number of CPU cores a virtual machine will be able to utilize on the host. This is done through the XML tag *cputune*:

```
  <cputune>
    <vcpupin vcpu='0' cpuset='0'/>
  </cputune>
```

Adding the lines above to the XML file specifies that virtual CPU 0 is to be ran by the physical CPU core 0 on the host machine. [3]

2.5 Control groups (cgroups)

cgroups is a linux kernel feature used to organize tasks and their children into hierarchical groups. This is implemented in order to better track resource usage. Through the usage of cgroups it is also possible to limit resources, in order to prioritize certain tasks.

The cgroups active on a computer can be found within the */sys/fs/cgroup/* folder. There are several subsystems which handles different operations. There are isolation and special controllers, which include the cpuset, freezer, devices and checkpoint/restart subsystems. Additionally there is resource control which consists of cpu, cpuacct, memory, disk I/O and network. By adding tasks to these subsystems it is possible to control different aspects of the task.

In order to manage cgroups, *cgroups-bin* can be installed through apt-get. Then cgroups can be created through the *cgcreate* command. Tasks can then be added into the cgroup with the *cgclassify* command. When a cgroup is created it is possible to impose restrictions on the tasks within it by altering values within specific files.

The following lines will create a cgroup, add some task to the cgroup, and finally change the priority the task will get on the CPU to 2048 from the default of 1024:

```
# cgcreate -g cpu:/<some group>
# cgclassify -g cpu:/<some group> <PIDs of tasks to put in cgroup>
# echo 2048 > /sys/fs/cgroups/cpu/'some_group'/cpu.shares
```

Virtual machines created through libvirt can be found in folders within the different subsystems. Within the file *cgroup.procs* found in the different cgroups, one can see which tasks belong to the cgroup. It is also possible to go the other way around, and find the hierarchy for a task within the file */proc/<PID of the task>/cgroup* [16, 17, 18]

2.6 Non-Uniform Memory Access (NUMA)

NUMA is an architecture designed to scale well with a large amount of CPUs. Most desktop computers use a symmetric multiprocessing architecture (SMP). All CPU cores share front-side bus and memory bus, as shown in figure 2.1. This works well when the number of CPU cores is relatively low. However, as systems get a larger amount of CPU cores the bus can become a bottleneck. An SMP architecture suffers from scalability limitations, and in order to overcome these limitations the NUMA architecture was designed.

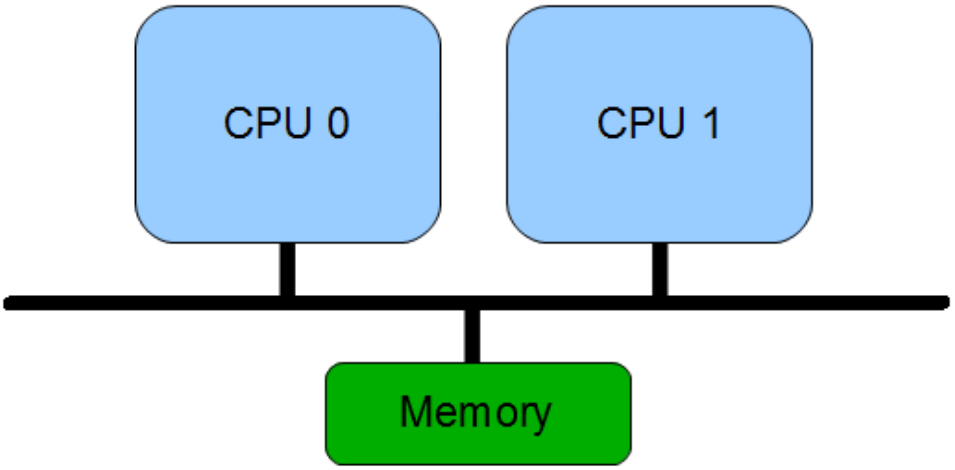

Figure 2.1: *SMP architecture with 2 CPUs*

With a NUMA architecture CPU cores are connected to different NUMA nodes, while the CPUs are connected by a high speed interconnection, as shown in figure 2.2. This architecture minimizes the issue with the memory bus being a bottleneck, but it does introduces some new concepts.

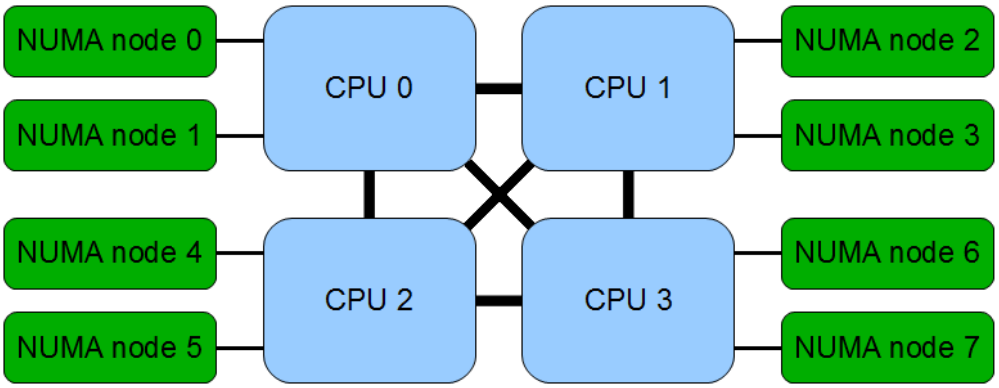

Figure 2.2: *A 4-socket AMD NUMA node architecture*

A machine with a NUMA architecture will have local and remote memory. A CPU core using the memory node it is directly connected to is using the local memory, while a CPU core using memory it is not directly connected to is using remote memory.

These new concepts can cause some issues if the software running on the machine is not aware of the architecture. One issue that can occur is lock starvation. A common implementation of memory locking is to let processes constantly check to see when a lock becomes available. The first one to to check after it has become available will be able to request a new lock. With a NUMA system the processes on the CPU connected to the NUMA node will have an advantage in obtaining new locks, due to the lower latency. This can lead to processes on CPUs trying to use the remote memory being locked out by the CPUs using their local memory.

It is possible to retrieve information about the NUMA architecture with the *numactl* command. When ran on the high-performance server used in this project, it produces the following results:

```
# numactl --hardware
available: 8 nodes (0-7)
node 0 cpus: 0 4 8 12 16 20
node 0 size: 16374 MB
node 0 free: 15932 MB
node 1 cpus: 24 28 32 36 40 44
node 1 size: 16384 MB
node 1 free: 16015 MB
node 2 cpus: 2 6 10 14 18 22
node 2 size: 16384 MB
node 2 free: 15918 MB
node 3 cpus: 26 30 34 38 42 46
node 3 size: 16384 MB
node 3 free: 16021 MB
node 4 cpus: 3 7 11 15 19 23
node 4 size: 16384 MB
```

```
node 4 free: 15884 MB
node 5 cpus: 27 31 35 39 43 47
node 5 size: 16384 MB
node 5 free: 16013 MB
node 6 cpus: 1 5 9 13 17 21
node 6 size: 16384 MB
node 6 free: 16015 MB
node 7 cpus: 25 29 33 37 41 45
node 7 size: 16368 MB
node 7 free: 16003 MB
node distances:
node   0   1   2   3   4   5   6   7
  0:  10  16  16  22  16  22  16  22
  1:  16  10  22  16  16  22  22  16
  2:  16  22  10  16  16  16  16  16
  3:  22  16  16  10  16  16  22  22
  4:  16  16  16  16  10  16  16  22
  5:  22  22  16  16  16  10  22  16
  6:  16  22  16  22  16  22  10  16
  7:  22  16  16  22  22  16  16  10
```

This tells us that the machine has 8 numa nodes of 16 GB. The last matrix describes the distance between the NUMA nodes. [19, 20, 21, 22]

Chapter 3

Approach

3.1 Methodology

In order to find out more about what causes the performance issues when creating large numbers of virtual machines through libvirt, multiple methods will be used. This will hopefully provide a broad understanding of the issue, what causes it, and how it can be improved upon.

Scalability experiments

Tests will be performed were the goal is to set up many small virtual machines under different variables. This is needed in order to confirm that there is an issue, as well as showing which parameters may affect the performance of the host machine. It may also directly help in figuring out what causes the issue. As well as providing information, the experiments may also provide a set of parameters which allows for large numbers of virtual machines to be ran efficiently.

Contact with the development team

Once the issue has been confirmed and investigated through the experiments, the development team will be contacted. They might be interested in the results, and they may have more knowledge about the problem.

Hypothesis testing in a reversed experiments

An experiment will be performed trying to replicate the results from the libvirt experiments while using virtual machines created through QEMU commands. libvirt creates cgroups for every virtual machine that is created, while virtual machiness created through QEMU commands will not be put in their own separate cgroup. The mail from the libvirt developers suggested that the indivitual cgroups might be causing the issue. By either turning of this functionality in libvirt or adding this functionality to virtual machines created through QEMU commands it is possible to determine if cgroups really is an issue.

Profiling

After obtaining more information about the issue, profiling will be performed in order to find out more about the underlying causes. This will hopefully give information about what specific functions which consumes the hosts resources.

Updated software

The libvirt developers suggested that the performance issue had for the most part been solved in the latest kernel versions. A test with an updated version of libvirt and the linux kernel will be performed to see if this is the case. Additionally profiling will be performed on the updated software.

3.2 Test machines

The host machines will be using 64-bit Ubuntu 12.04.1 LTS Server for the scalability experiments. This version ships with kernel version 3.2.0-23.36. The libvirt and QEMU versions used are the ones default to the system, QEMU version 1.0 and libvirt version 0.9.8.

For the profiling a 64-bit Ubuntu 12.10 Server installation will be used. This version comes with kernel version 3.5.0-17-generic. It will by default install libvirt version 0.9.13 and QEMU version 1.2.0. While this change is not optimal it was done because multiple people were using the high-performance system and an upgrade was required by another user. Tests were performed which showed that the same issues was still present with newer versions of the software. There should be no problem involved with performing the profiling on newer versions of the software, as the same performance issues still persisted.

System specifications Two machines will be used for experiments and work in this thesis. A regular desktop machine with a dual-core processor and a high-performance server with a massively multi-core processor.

Dell OptiPlex 755:

- Intel Core 2 Duo CPU E6550
- 4096MB DDR2 SDRAM

Dell PowerEdge R815:

- 4x AMD Opteron Processor 6234
- 16x 8192MB DDR3 SDRAM

3.3 Installation

Some packages and prerequisites are required in order to start using libvirt with KVM.

Enable KVM support In order to find out if KVM is supported on a processor, check the /proc/cpuinfo file with the following command:

```
# egrep -c '(vmx|svm)' /proc/cpuinfo
```

1 or more indicates that hardware virtualization is supported. Even though the processor supports KVM, it must also be enabled in the BIOS. Look for a way to enable Intel VT or AMD-V, depending on the CPU type. This is turned off by default, so it is likely that this will require a change.

Installing libvirt and QEMU For a hosts to be able to run QEMU virtual machines some packages and configurations are needed. The packages can be installed through apt-get:

```
# sudo apt-get install qemu-kvm libvirt-bin bridge-utils
```

libvirt-bin is the package that provides the libvirt API, qemu-kvm provides the backend which is needed for creating QEMU virtual machines while bridge-utils provides a bridge between your network and the virtual machines.

For a user to be able to use libvirt commands, it must be a part of the libvirtd group:

```
# sudo adduser <username> libvirtd
```

This command will first take effect once the user logs into the system, so relogging onto the system will be needed. With these configurations done the host machine should be ready to run KVM supported virtual machines [23].

Perl installation On a default Ubuntu installation perl is available. However, the scripts used for the experiments uses some additional Perl modules. Sys::Virt is easiest to install through apt-get:

```
# sudo apt-get install libsys-virt-perl
```

Data::UUID can be installed through CPAN. The CPAN installation procedure will ask some questions about how to perform the installation. Using the default setting for all of them will work.

```
# sudo cpan Data::UUID
```

3.4 Scalability experiments

In order to learn more about the scalability issues a set of experiments will be performed in order to measure resource usage under different scenarios.

3.4.1 Scripts

Perl scripts were made in order to perform the tests. A simple script was created, and small variations were made within the XML section of the script in order to make the script test different variables. One variation of the script can be found in the appendix A. The script goes through a loop in which it creates a new virtual machine, runs vmstat for a short while in order to measure various information about the system and then writes this information to a file. The name of the file and the start and stop points for the loop is specified through command line arguments. The file written contains all data from a vmstat run, along with the epoch time of when it was created, as well as the UUID of the virtual machine and a confirmation that the virtual machine was successfully started.

3.4.2 Disabling KVM

Experiments will also be performed with and without KVM. In order to disable KVM until the machine is restarted, it is possible to unload the module. The command will vary based on the processor type, either use:

```
# rmmod kvm-intel
```

or:

```
# rmmod kvm-amd
```

3.4.3 ulimit

When performing initial tests it became clear that the systems performance was not the limiting factor when it came to creating a large amount of virtual machines. Default limitations within Ubuntu prevents users and processes from opening more than 1024 files. However, this can be controlled with the ulimit command:

```
# ulimit -Hn
# ulimit -Sn

# ulimit -Hn 2048
# ulimit -Sn 2048
```

The n-option controls the number of open file descriptors, while the H- and S-option lets you change or report the soft and hard limits. The soft limit is what actually decides how many files which may be opened, while the hard limit decides the maximum value for the soft limit. [24]

The first two commands above will show the limits for the user logged on while the last two will change the values.

In order to increase the amount of files libvirt is able to open changes must be made to */etc/default/libvirt-bin*. By adding the ulimit commands for changing of the hard and soft limits into this file the libvirt daemon will be able to open more files once it has been restarted. [25]

3.4.4 Variables to test

Quite a few combinations of settings will be tested. Two different computers will be used for the experiments, one simple desktop with an Intel Dual-Core and one high-end server with a massively multi-core AMD processor.

On the desktop tests will be ran while using both CPU cores and while only using one of them. Runs will also be performed were the virtual machines are created using QEMU commands without libvirt. The experiments with libvirt will be performed both with and without KVM enabled

On the high-performance server tests will be ran with manually setting the virtual machines to run on 1, 10 and all 48 of the cores. Two types of tests will be performed while using all cores. One where the cores used is manually set, and one where nothing is specified. All of these tests will also performed with and without KVM enabled. Finally runs will be performed where virtual machines are created using QEMU commands, without using libvirt.

This gives us the following experiments:

- desktop with KVM, no CPU-management
- desktop with KVM, only one CPU core used
- desktop without KVM, no CPU-management
- desktop without KVM, only one CPU core used
- desktop with KVM, without libvirt
- high-end server with KVM, no CPU-management
- high-end server with KVM, 48 CPU cores used
- high-end server with KVM, 10 CPU cores used
- high-end server with KVM, 1 CPU cores used
- high-end server without KVM, no CPU-management
- high-end server without KVM, 48 CPU cores used
- high-end server without KVM, 10 CPU cores used
- high-end server without KVM, 1 CPU cores used
- high-end server with KVM, without libvirt

Due to time restraints only 3 runs of each experiment will be performed.

3.5 Contact with the development team

With results gathered from the experiments, the developers will be contacted and asked what might be the cause of the poor performance. They are available through mailing lists as well as IRC, and can hopefully they can provide valuable input.

3.6 Hypothesis testing in a reversed experiment

In the-mail received prior to starting this project, as shown in appendix D, the libvirt developers suggested that cgroups might be causing the performance problems. No way of turning the cgroups completely off was found, so the other option was to add the cgroup functionality to virtual machines created through QEMU commands.

If virtual machines created through QEMU commands can be created with good performance and we only introduce one new variable, then we can see how this variable affects the performance.

If putting virtual machines created through QEMU commands into cgroups results in the same poor performance as observed when creating virtual machines through libvirt, then it is likely that cgroups is the issue.

The script used earlier will be modified to use the *cgcreate* and *cgclassify* commands to manually put each virtual machine into their own cgroup. This modified script can be found in appendix B.

3.7 Profiling

In order to figure out what causes the perfomance issues, two types of profiling will be performed.

3.7.1 Devel::NYTProf

Profiling will be performed on the perl code with Devel::NYTProf. This profiler can be installed through CPAN:

```
# cpan Devel::NYTProf
```

In order to perform profiling with NYTProf, the program which is to be profiled is ran with NYTProf:

```
# perl -d:NYTProf <program_to_profile>
```

When the program has finished running a nytprof.out file will have been created. This file can then be converted to a set of HTML files:

```
# nytprofhtml
```

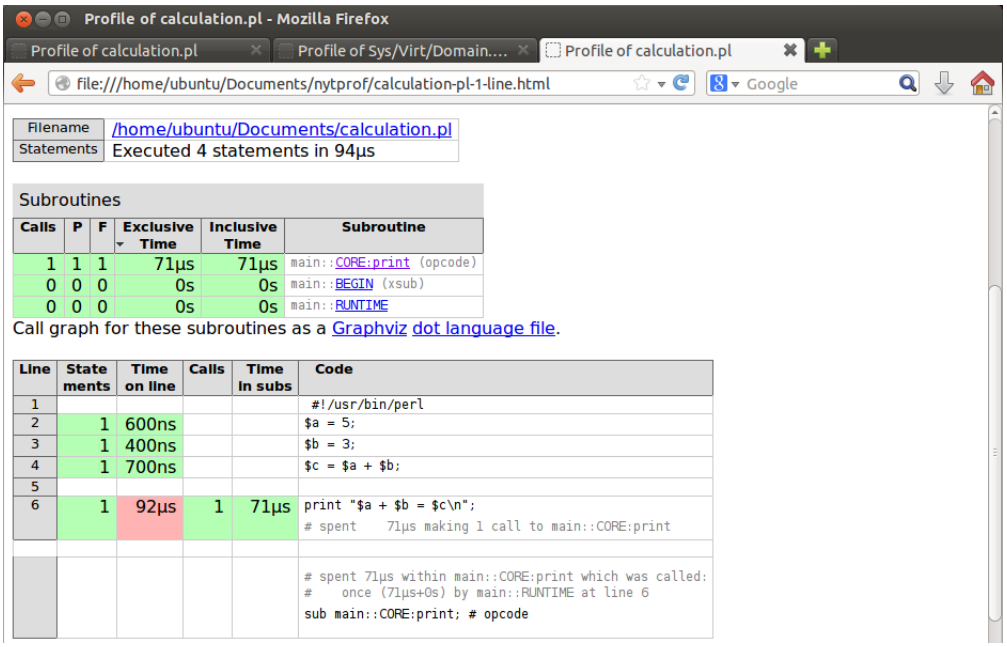

Figure 3.1: *A sample html page showing results from NYTProf profiling*

This gives a great way of viewing statistics of the program that has been executed, as seen in figure 3.1. The webpages produced give a well organized view of where time has been spent and are easy to navigate.

3.7.2 OProfile

OProfile can not be installed through apt-get, so it needs to be compiled from the provided tar.gz files on the oprofile website. Before this can be done, some dependencies must be installed:

```
# apt-get install libpopt-dev binutils-dev
```

Then a user and group named oprofile needs to be created, and everything should be ready for the installation. Unpack the .tar.gz file which can be downloaded from the oprofile pages, and perform the installation process:

```
# wget http://prdownloads.sourceforge.net/oprofile
    /oprofile-0.9.8.tar.gz
# tar -xzf oprofile-0.9.8.tar.gz
# cd oprofile-0.9.8
# ./configure
# make
# make install
```

This is all that is required for OProfile to work, but it will not be able to see what happens within the kernel. Whatever happens within the kernel will simply be listed as */no-vmlinux*. In order to get further information,

debug symbols for the kernel must be provided. These can be downloaded through apt-get, but requires some additional repositories. These repositories are added by appending the following lines to */etc/apt/sources.list.d/ddebs.list*:

```
deb http://ddebs.ubuntu.com quantal main restricted universe
    multiverse
deb http://ddebs.ubuntu.com quantal-updates main restricted
    universe multiverse
deb http://ddebs.ubuntu.com quantal-security main restricted
    universe multiverse
deb http://ddebs.ubuntu.com quantal-proposed main restricted
    universe multiverse
```

If Ubuntu 12.10 is not used, *quantal* should be replaced with the code name for the Ubuntu version in use.

After adding the repositores the packaging tool must be updated, and then the debug symbols can be downloaded:

```
# apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 428D7C01
# apt-get update
# apt-get install linux-image-$(uname -r)-dbgsym
```

The command *uname -r* will provide the the name of the current kernel. By using this command inside the apt-get command the correct version of the debug symbols will be downloaded.

After running these commands OProfile should be ready for use, and be able to provide detailed information about what is using the CPU time:

```
# operf --vmlinux=/usr/lib/debug/boot/vmlinux-$(uname -r)
    ./<program_to_profile>
# opreport  -l ./<program_to_profile>
```

Run a program with operf, and provide the path to the recently downloaded debug symbols. After running the program a report will be created, and this can be viewed through the *opreport* command.

In order to perform this profiling a C program was created in order to limit the number of variables, as shown in appendix C. Perl will communicate with the libvirt C code, but when using a C program the library for communication between Perl and C will not be needed. The C program can be found in appenix B. This code simply creates virtual machines, but does not spend time measuring resource usage.

If profiling is done throughout the creation of all of the virtual machines it will only give averages throughout the whole process. In order to see what is happening at different stages throughout the creation process profiling will be performed at certain intervals. Multiple versions of the C program will be ran in a sequence by a simple bash script. First a version creating 10 virtual machines will be ran while profiling is performed. Then a version creating 90 virtual machines will be ran without profiling. This will be repeated several

times, giving profiling data for the first 10 virtual machines created, the 101st to the 110th virtual machines created, the 201st to the 210th virtual machines created and so on.

3.8 Updated software

Updating the software used might be one way of mitigating the performance issues which occurs with large numbers of virtual machines. Therefore, experiments and profiling will also be performed on a newer version of libvirt with an updated kernel version. With these updates one set of tests will be performed with KVM, without manually configuring which CPU cores to use. Profiling will be performed with OProfile, but without debug symbols, as these was not found for this newer kernel version.

3.8.1 libvirt 1.0.5

In order to get a newer version of libvirt on an Ubuntu server, a tarball from the libvirt website must be compiled. But in order to successfully compile this, quite a few prerequisites are required. These can be installed through apt-get:

```
# apt-get install gcc make pkg-config libxml2-dev libgnutls-dev
    libdevmapper-dev libcurl4-gnutls-dev python-dev
    libpciaccess-dev libnl-3-dev libnl-route-3-dev
```

After extracting the contents of the tarball from the libvirt website, there are also some directories which needs to be specified when running configure:

```
# ./configure --prefix=/usr --localstatedir=/var --sysconfdir=/etc
```

With the prerequisites installed and directories specified the rest of the installation is straightforward:

```
# make
# make install
```

3.8.2 linux kernel 3.9.1-saucy

In order to upgrade the linux kernel a set of .deb packages must be downloaded. These packages can be installed through dbkg. Finally the Grub bootloader must be updated, and the new kernel is ready for use.

```
# wget http://kernel.ubuntu.com/~kernel-ppa/mainline/v3.9.1-saucy
    /linux-headers-3.9.1-030901-generic_3.9.1-030901.201305080210
    _amd64.deb
# wget http://kernel.ubuntu.com/~kernel-ppa/mainline/v3.9.1-saucy
    /linux-headers-3.9.1-030901_3.9.1-030901.201305080210_all.deb
# wget http://kernel.ubuntu.com/~kernel-ppa/mainline/v3.9.1-saucy
    /linux-image-3.9.1-030901-generic_3.9.1-030901.201305080210
```

```
    _amd64.deb

# dpkg -i linux-image-3.9.1*.deb

# update-grub
```

When booting after installing a new a kernel, it is possible to choose which kernel version to use by selecting it under the advanced tab in the Grub menu.

Chapter 4

Results

4.1 Scalability experiments

The scalability experiments performed on both a desktop machine as well as a high-performance server produced large amounts of interesting data. While they did prove that the performance on the host machine was very poor under some circumstances, the results were not as expected.

4.1.1 Software limitations identified

On the desktop machine tests showed that a very large number of virtual machines could be created. At first there were some issues with default limitations set by the operating system. The libvirt-bin process will by default only support 1024 simultaneous open files. This limitation caused an error at 1002 virtual machines, because the process had too many open files.

As shown in the approach this setting was changed, but without the limitation to the number of open files for the libvirt process a new error appeared at 1004 virtual machines:

```
Error while creating domain:internal error
    invalid use of command API
```

This error does not provide much information about what is actually wrong. As it is quite vague, it was also hard to search for a solution to it. No solution to this problem was found until after the experiments had been carried out. Considering how performance issues can be seen way before 1000 virtual machines have been created, it was not crucial to be able to exceed this limitation.

4.1.2 Memory limitations identified

On the desktop machine the available memory was somewhat of a limiting factor. Emulating more devices, such as sound cards, video cards and CD-roms turned out to cause the virtual machines to use more memory. When trying to create machines with multiple emulated devices the host would first run out of free memory, then run out of swap space, and finally start killing of

previously created virtual machines in order to create new ones. This gave the following type of error messages:

```
Error while creating domain: internal error Child process

Out of memory: Kill process 1714 (kvm) score 1 or sacrifice child
Killed process 1714 (kvm) total-vm:340600kB, anon-rss:428kB,
    file-rss:92kB
```

This message tells that process 1714, which is a previously created virtual machine, has been killed in order to create a new one. By limiting the virtual machines to using few devices the desktop machine would run out of memory around 750 virtual machines, as seen in figure 4.1. Disabling KVM causes a little more memory usage, and the host machine would run out of memory at around 650 virtual machines. Memory used is very similar across all the tests. At some point the machine runs out of free memory and starts using the swap. When this happens, the time to create new virtual machines increases.

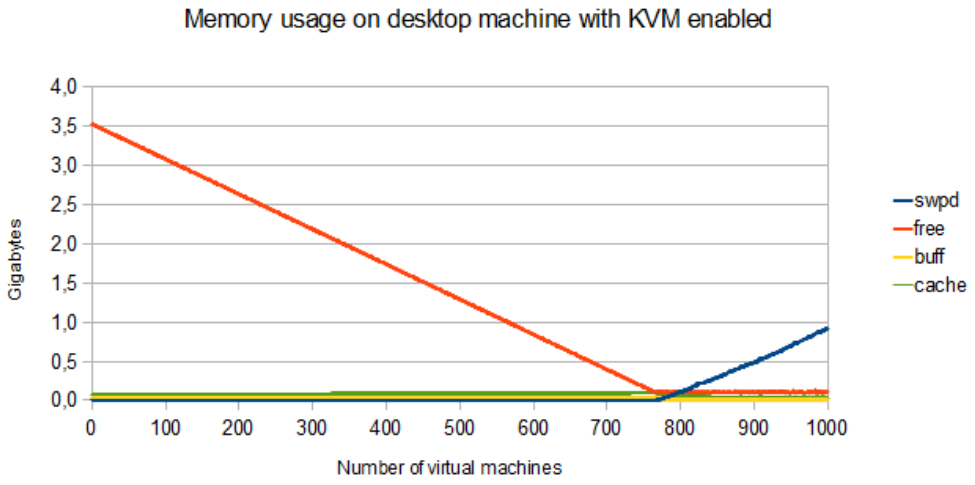

Figure 4.1: *Graph showing memory usage on the desktop machine as virtual machines are created*

On the high-performance server memory was never an issue. Even though more devices could have been emulated on the high-performance server without running into memory issues, the virtual machines were kept as minimal as possible in all of the tests.

4.1.3 Experiments on the desktop machine

When performing these experiments, the goal was to accurately determine the extent of the scalability limitations indicated by the preliminary tests.

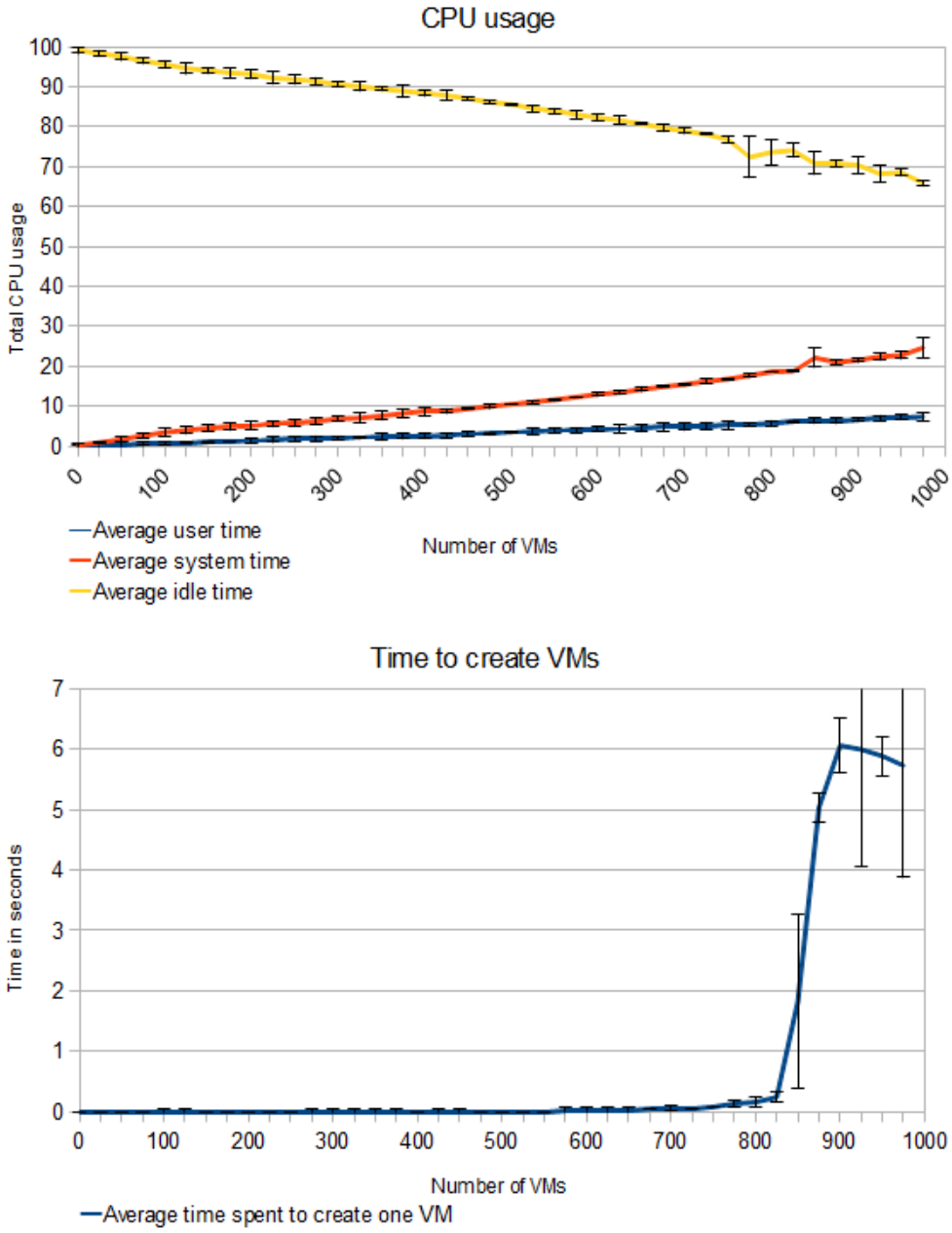

Figure 4.2: *CPU usage and creation time on the desktop machine with KVM enabled*

When creating virtual machines with libvirt on the desktop machine with KVM enabled the host machine performs relatively well, as seen in figure 4.2. The time create new virtual machines increases drastically at some point, but this is related to the host machine running out of free memory and starts using the swap space, as seen in 4.1.

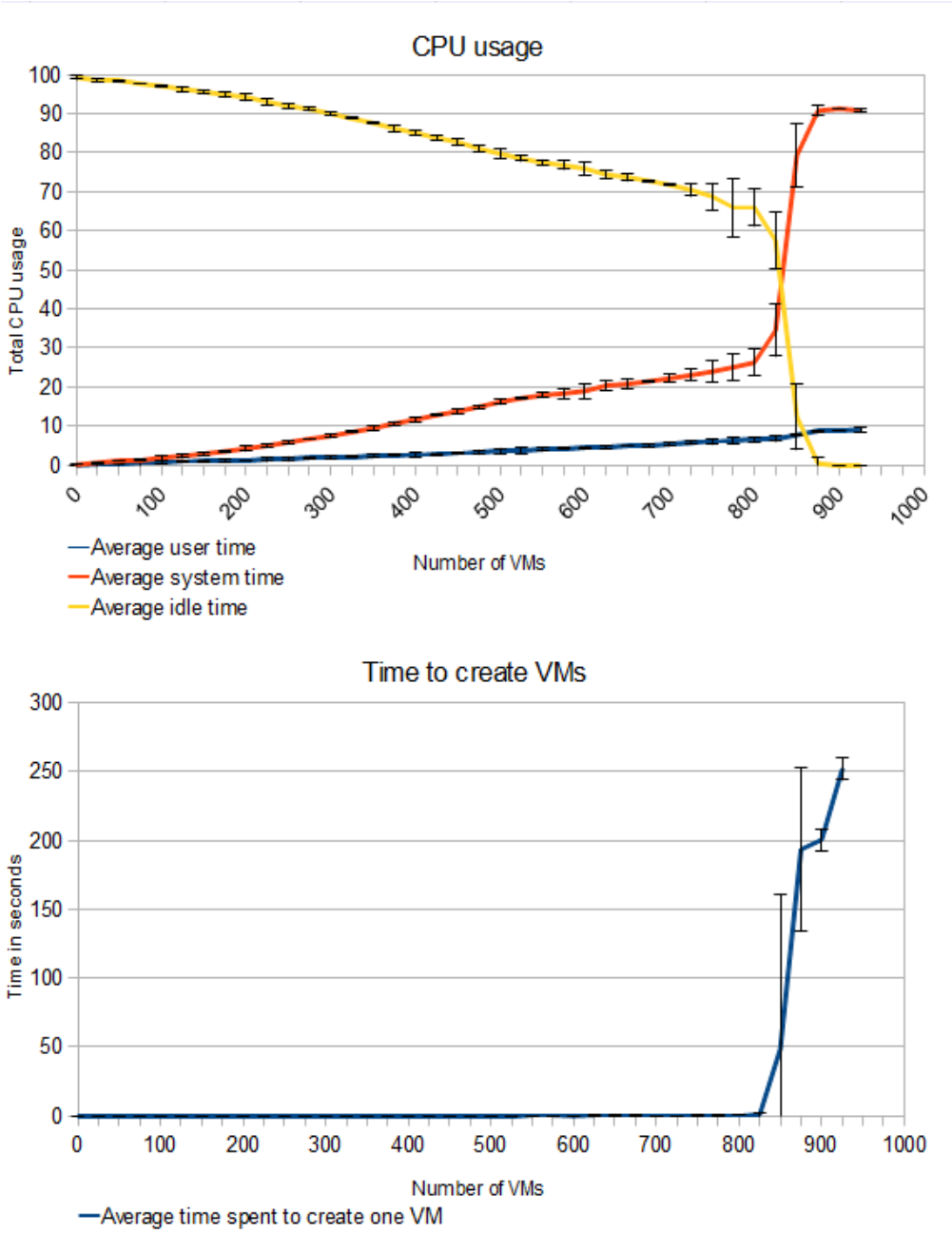

Figure 4.3: *CPU usage and creation time on the desktop machine with KVM enabled and only one CPU core being used*

When limited to using only one of the CPU cores, the performance gets worse, as seen in figure 4.3. At the same point as the host machine runs out of free memory, the CPU usage also rapidly increases. Even though only one CPU is being used for the virtual machines, all of the CPU's time gets used as it runs out of free memory.

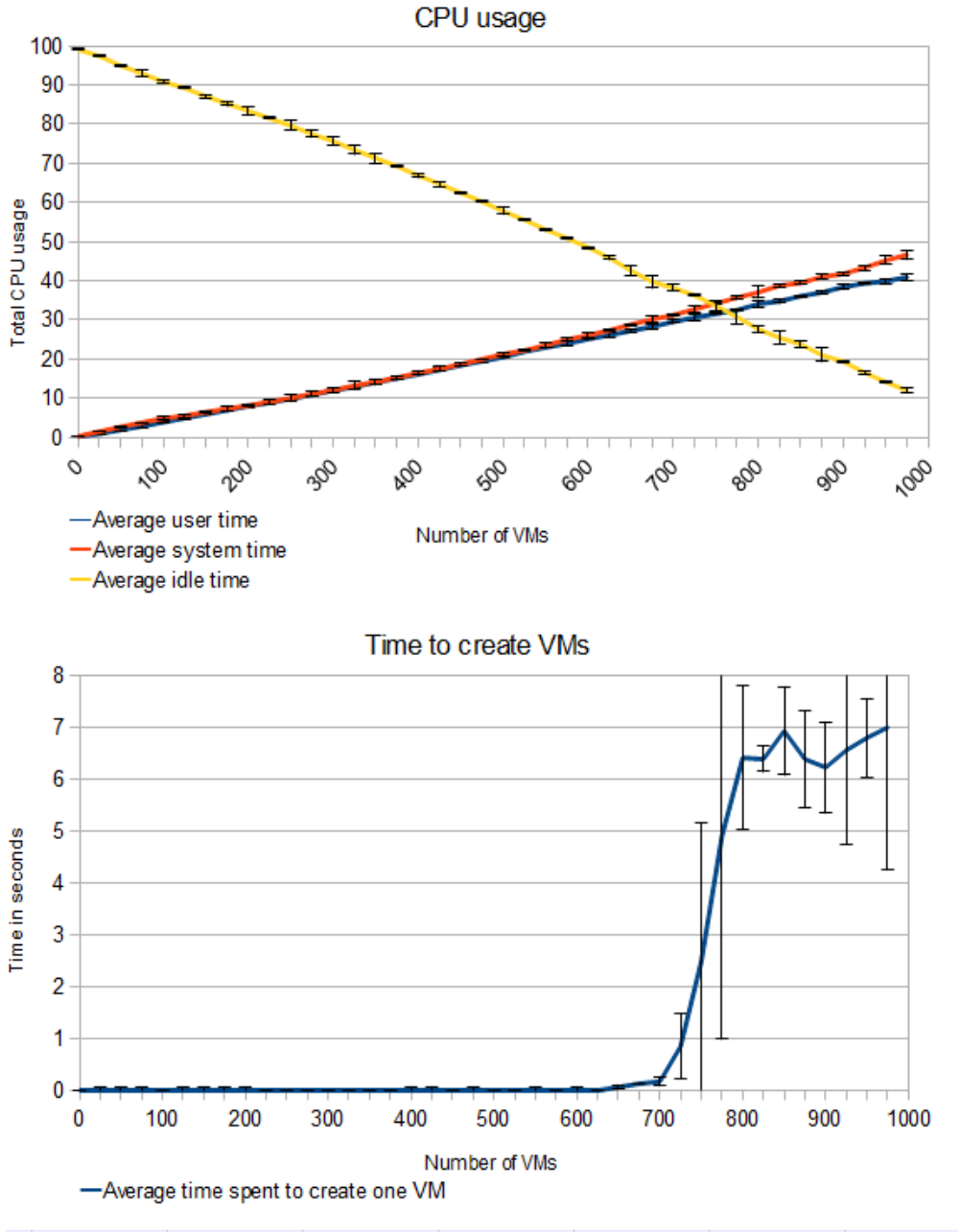

Figure 4.4: *CPU usage and creation time on the desktop machine with KVM disabled*

When disabling KVM, more CPU time is required, as seen in figure 4.4. Here the time spent on user time and system time increases about equally, which causes less CPU time spent idle. Disabling KVM causes slighty more memory usage, and as such the time to create new virtual machines increases at an earlier point.

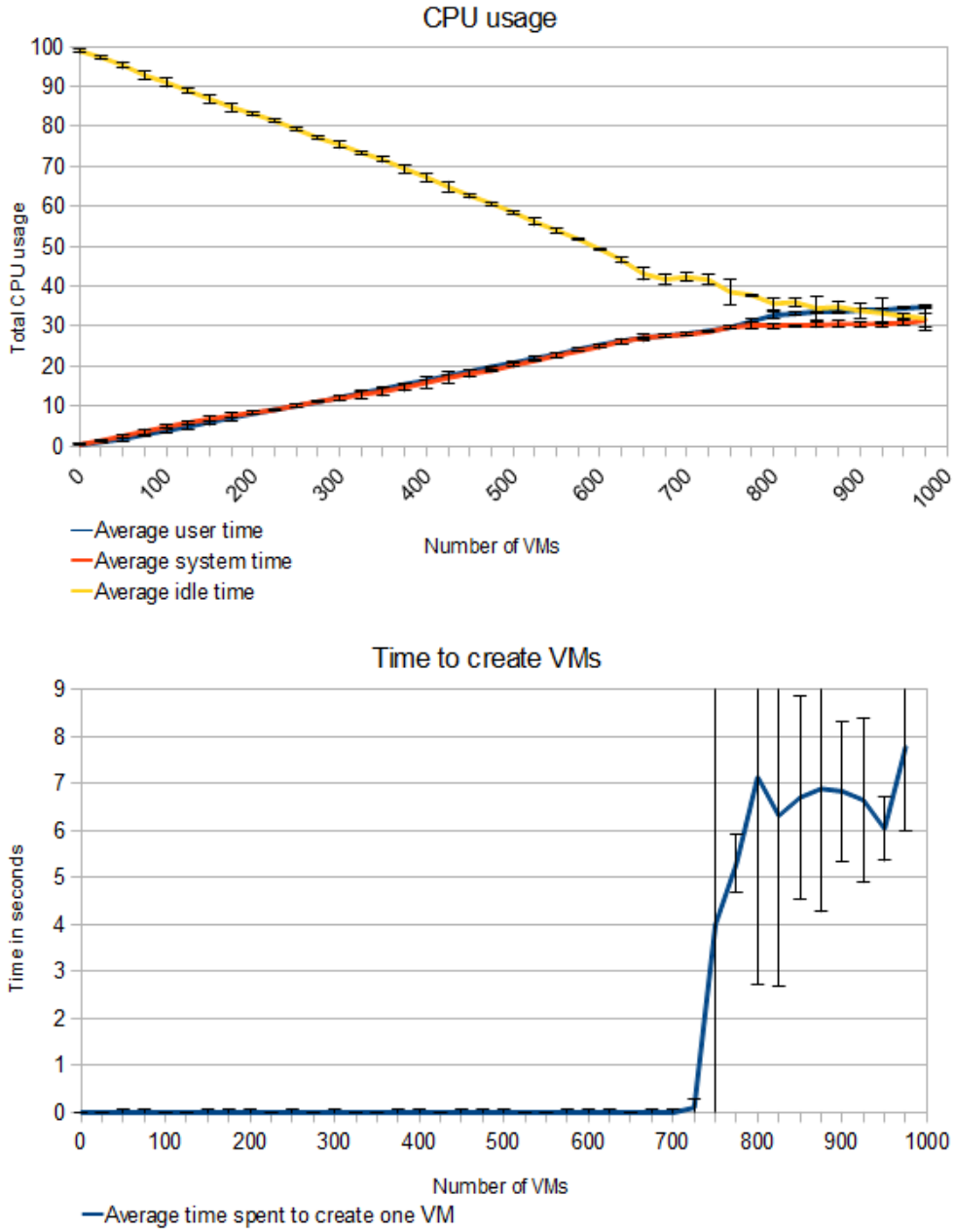

Figure 4.5: *CPU usage and creation time on the desktop machine with KVM disabled and only one CPU core being used*

With KVM disabled and only one CPU core being used the performance is still quite good, as seen in figure 4.5. More than 50% of the CPU time is being used, which shows that even though only one CPU core runs the virtual machines, both cores are affected. The error bars are quite large when it comes to the time it takes to create new virtual machines. This is because the slowdowns occur at slightly different times, as well as relatively large variations in

the time it takes to create new virtual machines. The trend that it takes more time after having created around 700 virtual machines is still clear.

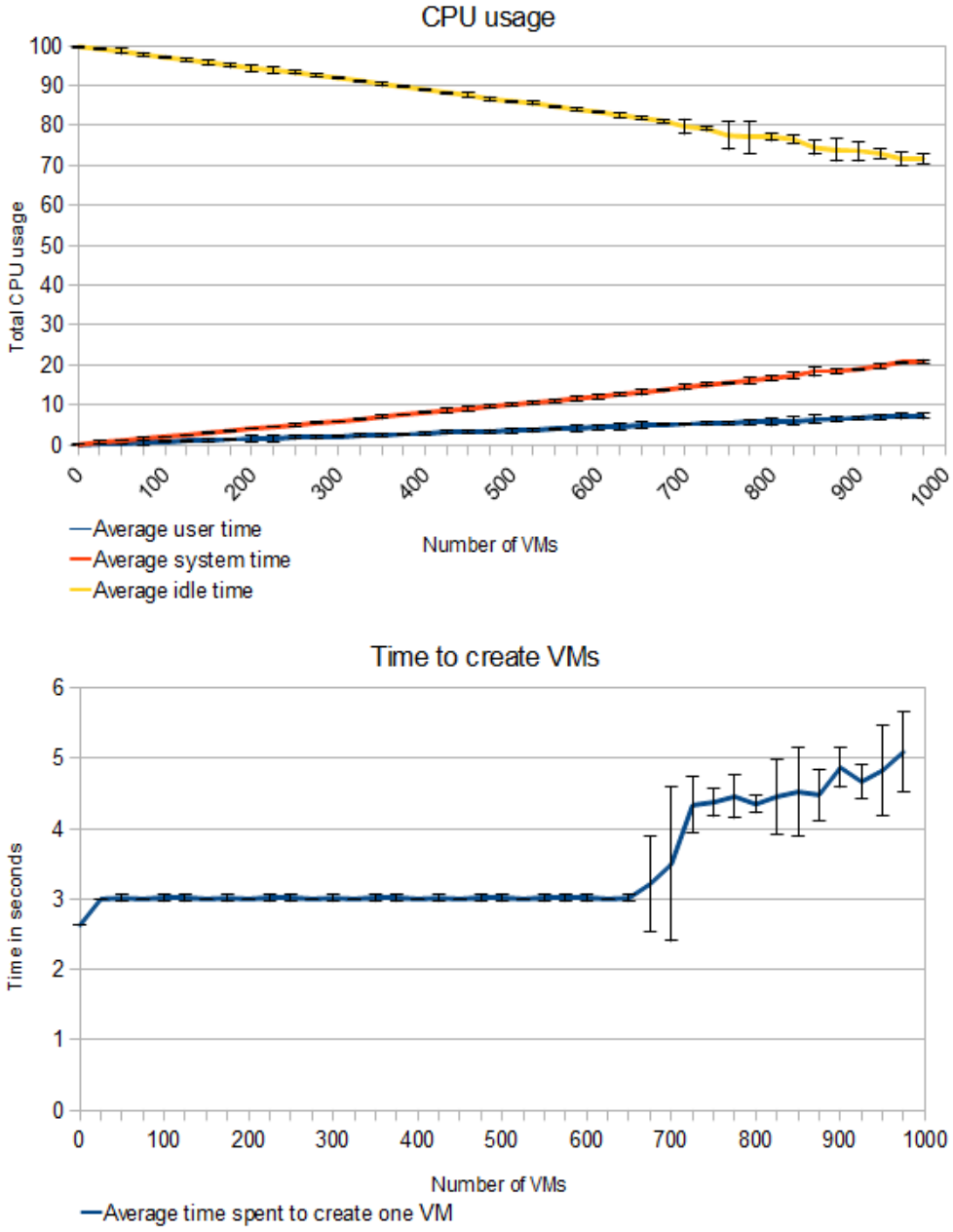

Figure 4.6: *CPU usage and creation time on the desktop machine with KVM enabled and VMs created through QEMU commands*

When creating virtual machines through QEMU commands, the CPU usage is very similar to that of creating virtual machines with libvirt, as seen when comparing figure 4.6 to 4.2. The time to create new virtual machines is a little longer with low numbers, but does not rise as much when the host

machine runs out of free memory

Summary

Overall the performance on the desktop machine is quite good. Limiting the virtual machines to only using one of the CPU cores decreases the performance, which is to be expected. There are only small performance differences when it comes to creating virtual machines through QEMU commands compared to creating them through libvirt.

4.1.4 Experiments on the high-performance server

It could be expected that the same experiments on a more powerful machine would yield better results, yet this was not the case.

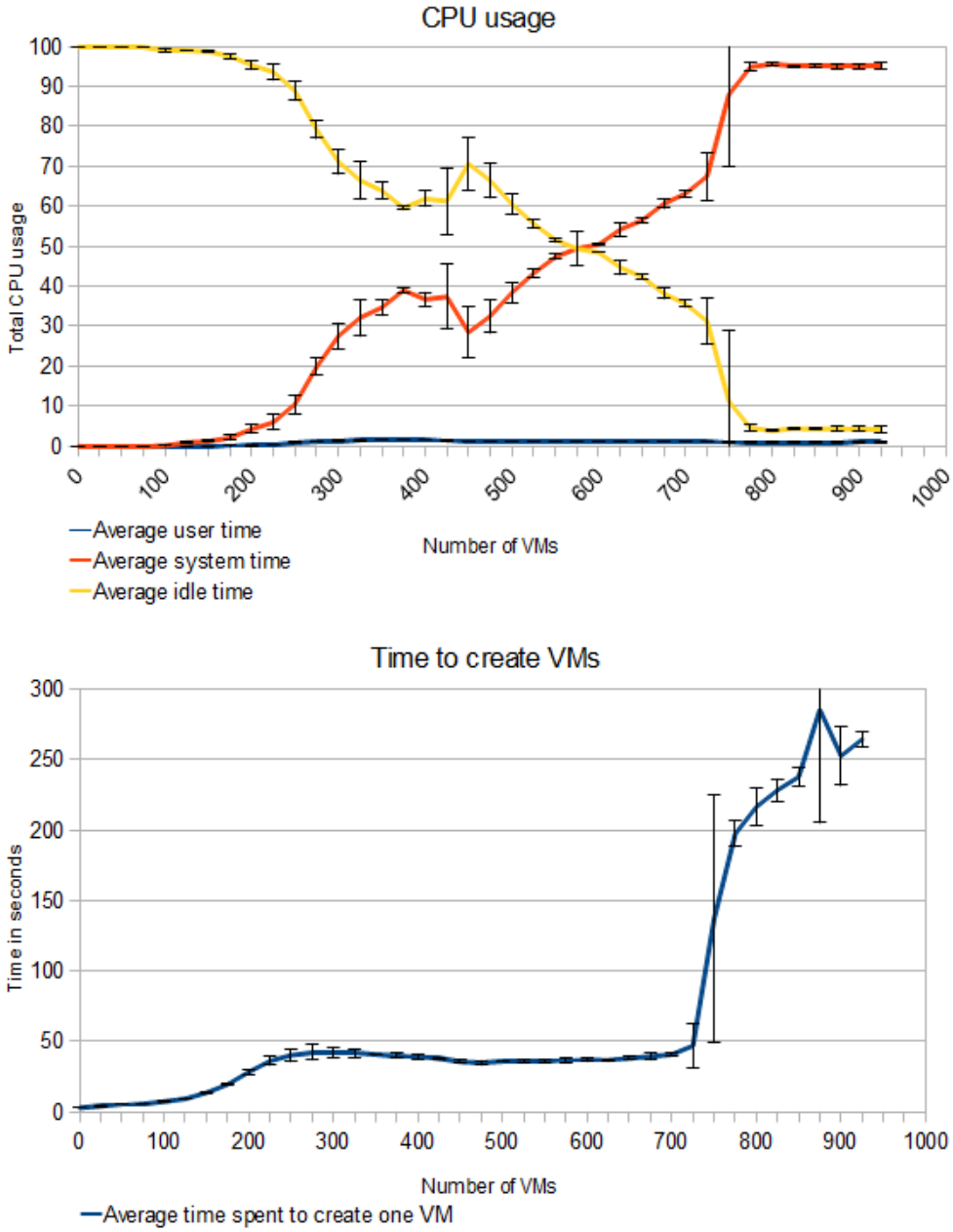

Figure 4.7: *CPU usage and creation time on the high-performance server with KVM enabled*

Using libvirt with KVM enabled while not specifying which CPU cores to use would be the standard way of setting up virtual machines. However, as shown in 4.7, this results in poor performance. After creating about 700 machines the CPU time spent on system time as well as the time to create new machines drastically increases. On this high-performance server there is still plenty of free memory.

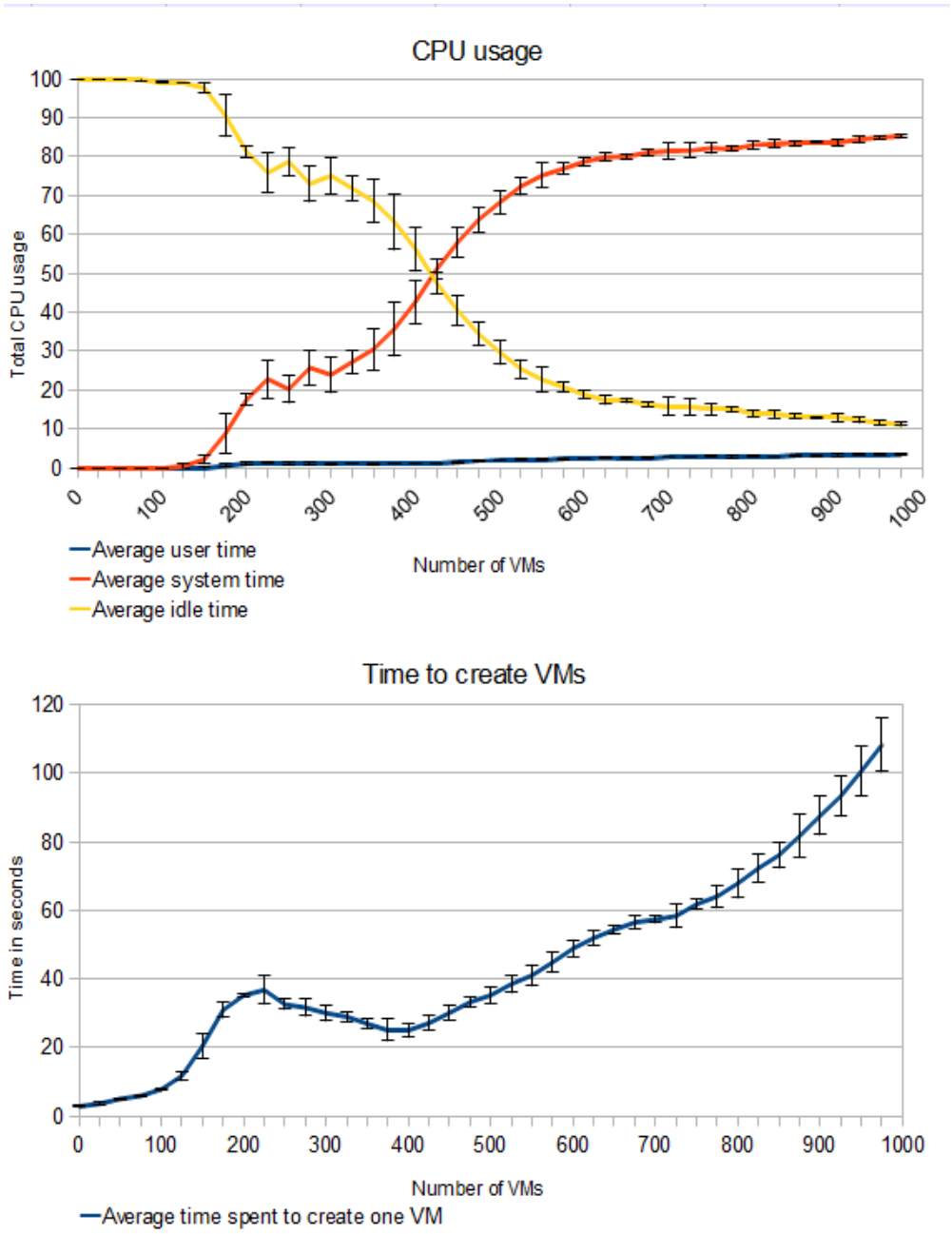

Figure 4.8: *CPU usage and creation time on the high-performance server with KVM enabled and a manually designated CPU core for each virtual machine*

When manually designating the CPU core for each virtual machine, as seen in figure 4.8 the sudden increase in CPU usage is not present. The system performs a little better as the number of virtual machines reaches a high number. The performance is still very bad considering the powerful hardware.

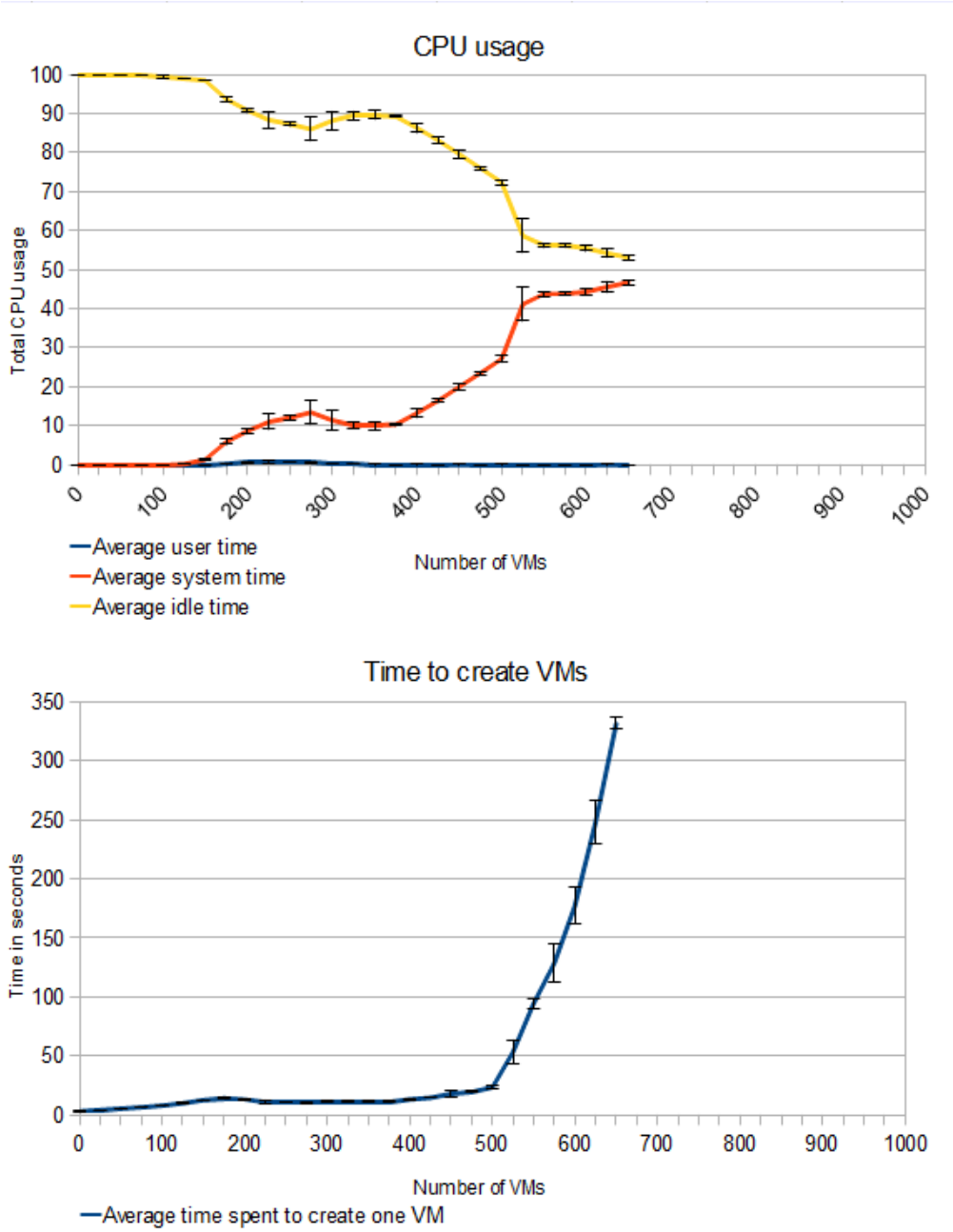

Figure 4.9: *CPU usage and creation time on the high-performance server with KVM enabled and only 10 CPU cores being used*

When only using 10 CPU cores the time to create new machines would rapidly increase after having created around 500 machines, as seen in figure 4.9. 10 CPU cores being used 100% should result in a 20.8% total CPU usage. The graphs show that more than 20.8% of the CPU time is being used. This shows that even though the virtual machines are set to specific CPU cores, other cores are still be affected. After having created around 650 virtual ma-

chines they would no longer reliably boot.

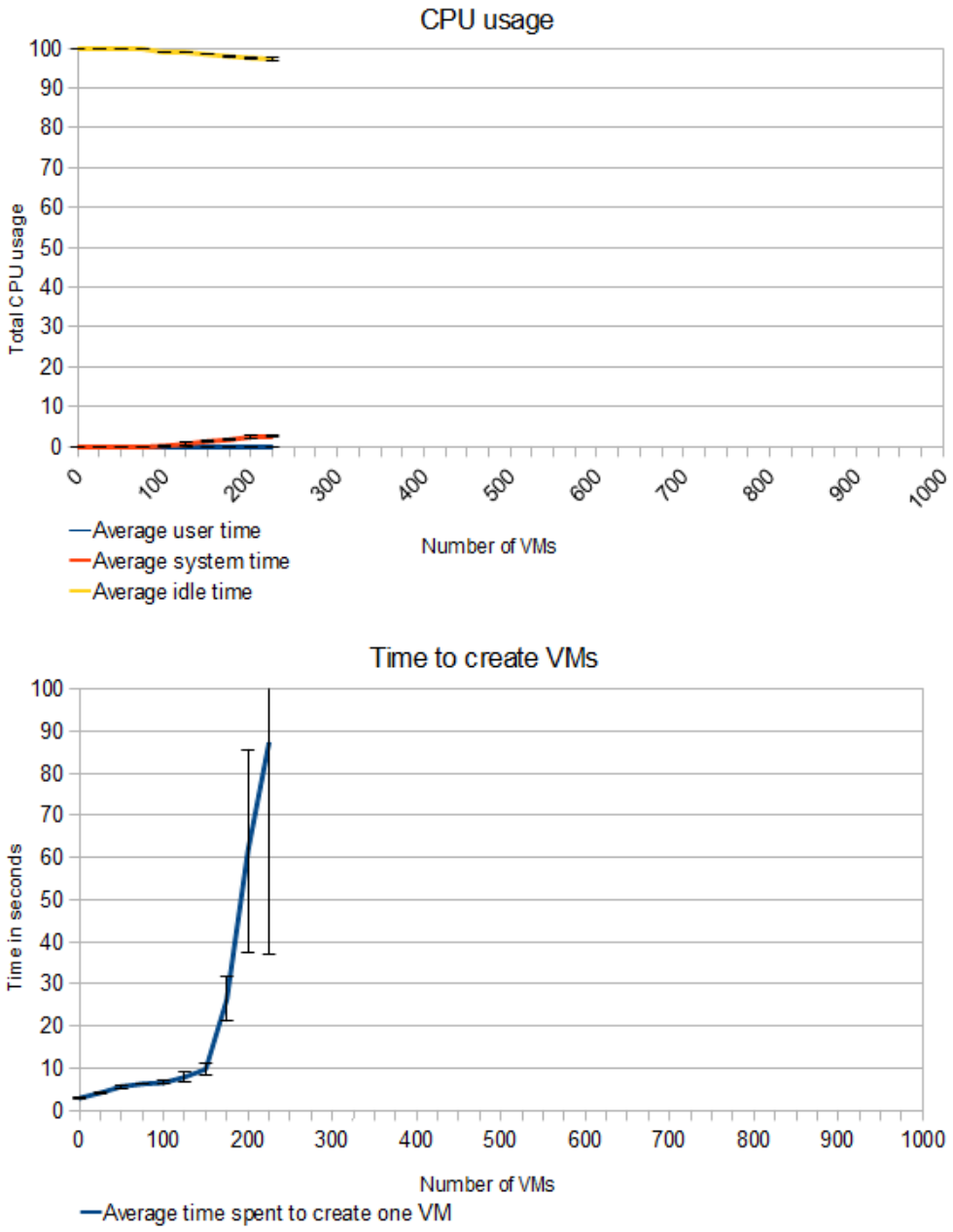

Figure 4.10: *CPU usage and creation time on the high-performance server with KVM enabled and only one CPU core being used*

When limiting the virtual machines to only use one out of the 48 available CPU cores, as seen in figure 4.10, the time to create new virtual machines would rapidly increase after having created around 150. A little while later the virtual machines would no longer reliably boot.

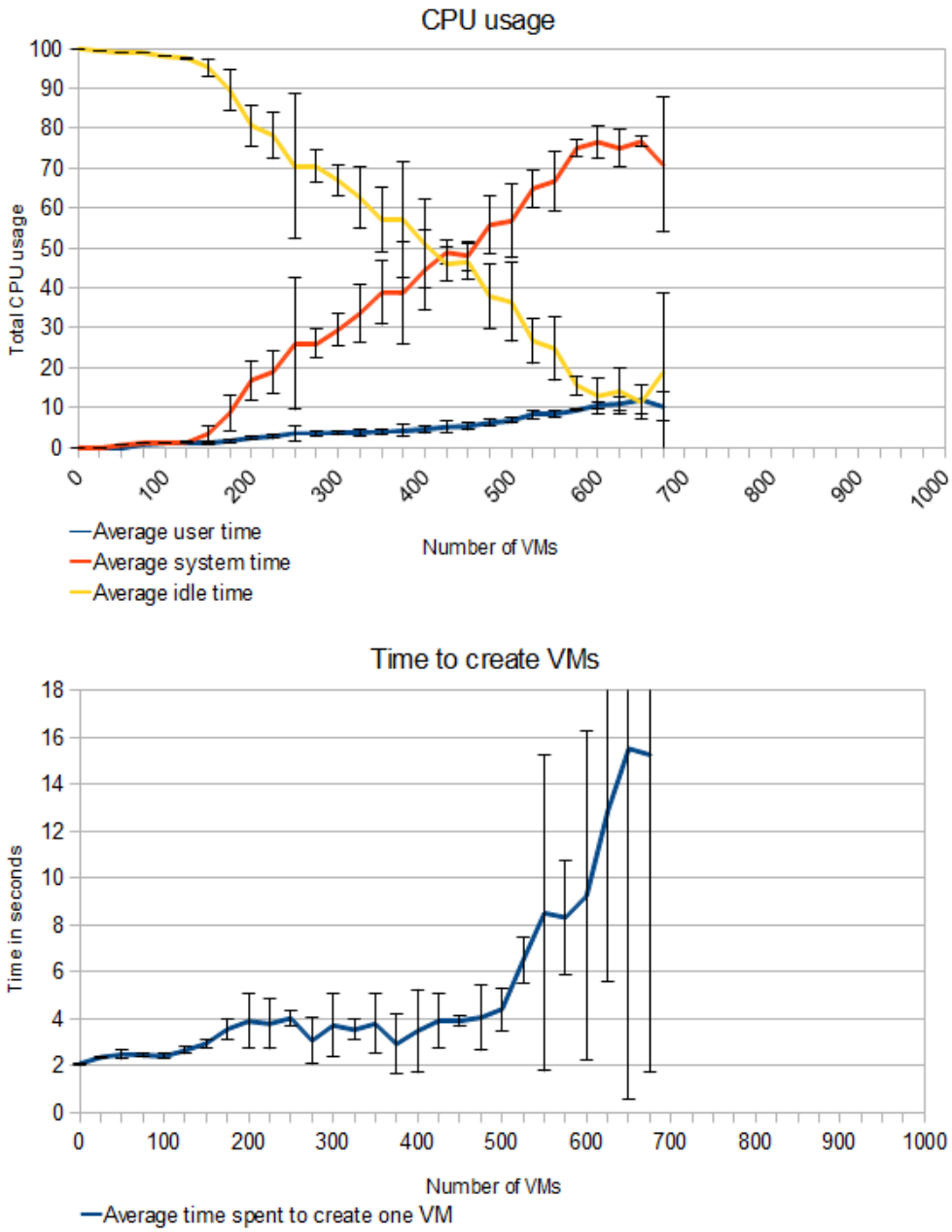

Figure 4.11: *CPU usage and creation time on the high-performance server with KVM disabled*

Turning off KVM did not increase the performance significantly, as seen in figure 4.11. While the CPU still spends some time idle, the virtual machines would no longer start reliably after reaching about 700 virtual machines. There are quite large variations between the different runs when it comes to the time it takes to create new machines, resulting in large error bars. However, the same general trend is seen across the different runs.

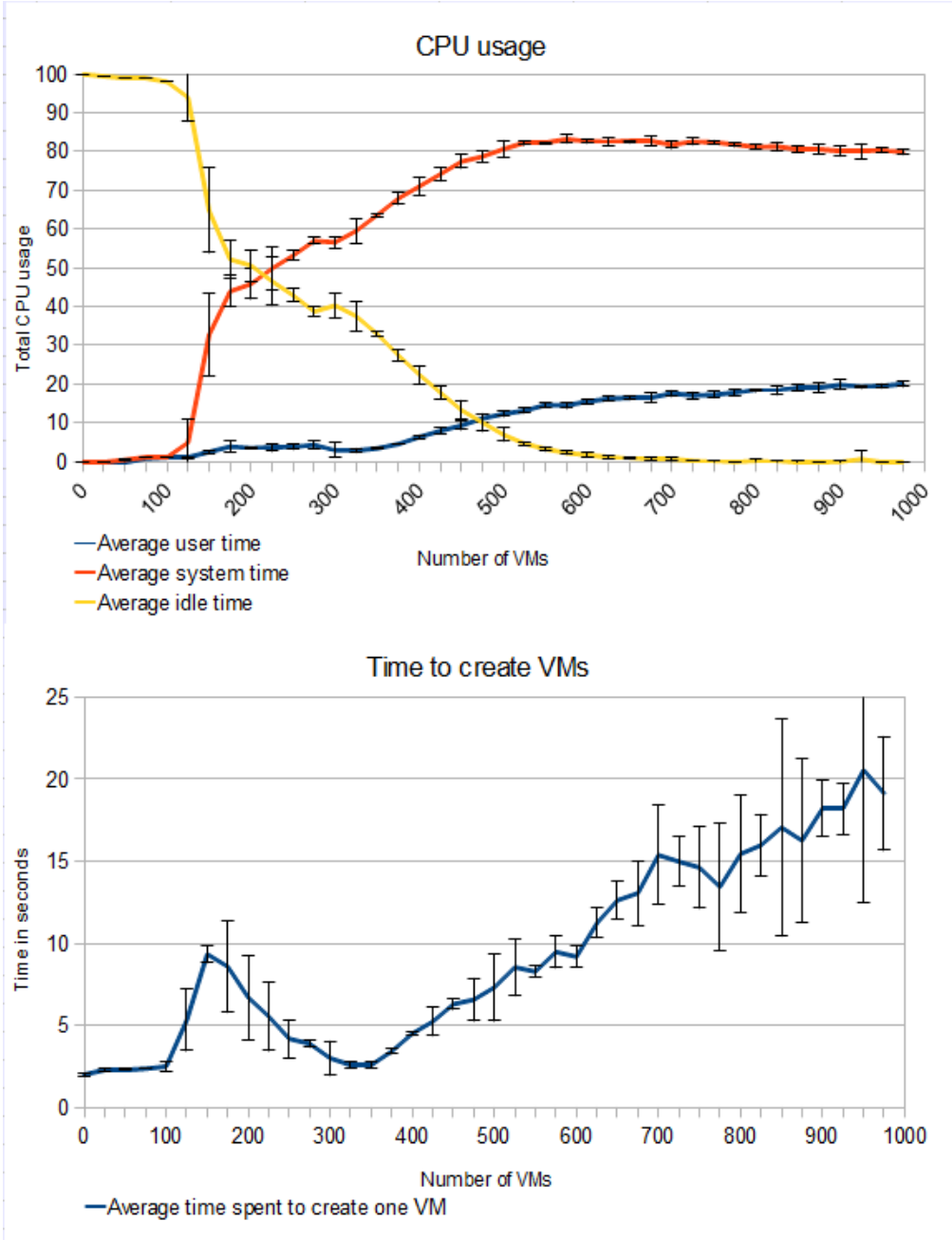

Figure 4.12: *CPU usage and creation time on the high-performance server with KVM disabled and a manually designated CPU core for each virtual machine*

When manually designating the CPU core for each virtual machine the overall CPU usage increases, as seen in figure 4.12. However, the virtual machines do reliably start.

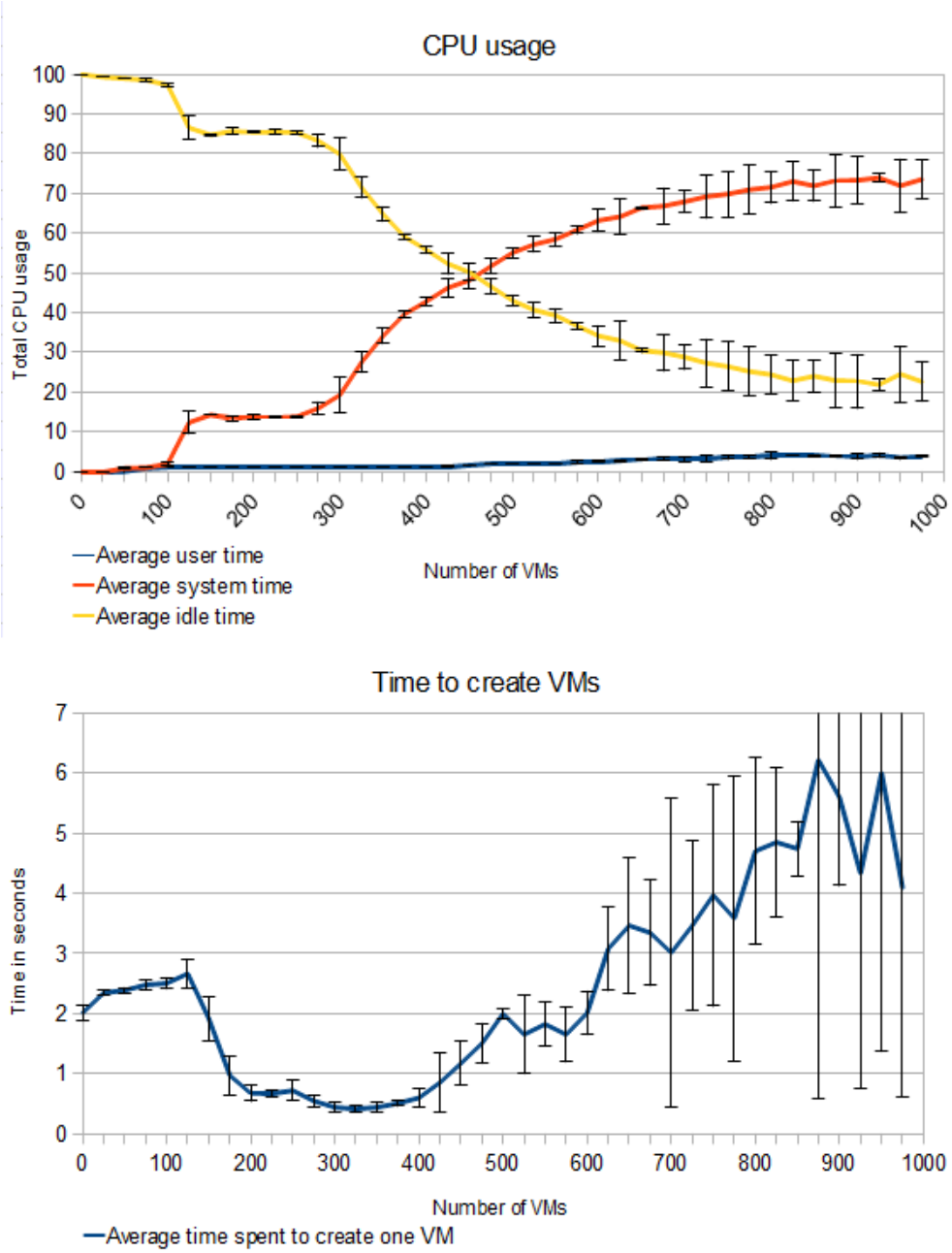

Figure 4.13: *CPU usage and creation time on the high-performance server with KVM disabled and only 10 CPU cores being used*

Limiting the virtual machines to only using 10 of the CPU cores results in lower overall CPU usage, as seen in figure 4.13. Similar to the test with 10 CPU cores and KVM enabled, the CPU usage exceeds 20.8%, which would equate to 10 cores being used 100%. The variations in the time to create virtual machines are quite large between the different runs, but overall it is much faster than the experiments where all CPU cores are being used.

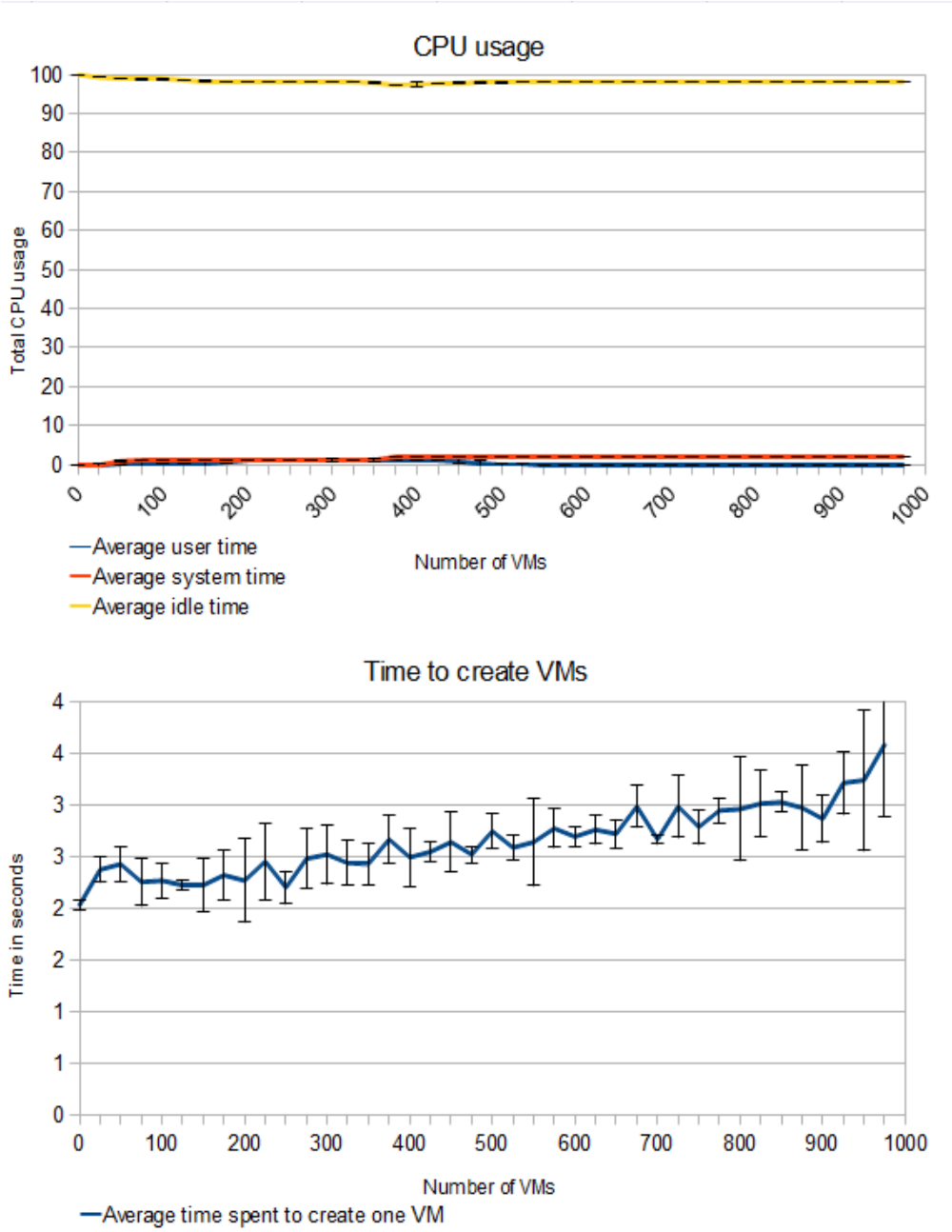

Figure 4.14: *CPU usage and creation time on the high-performance server with KVM disabled and only one CPU core being used*

When only using one CPU core the performance is quite good, as seen in figure 4.14. However, the one CPU core being used spends no time idle. While the results are quite good, the utilization of the hardware is not.

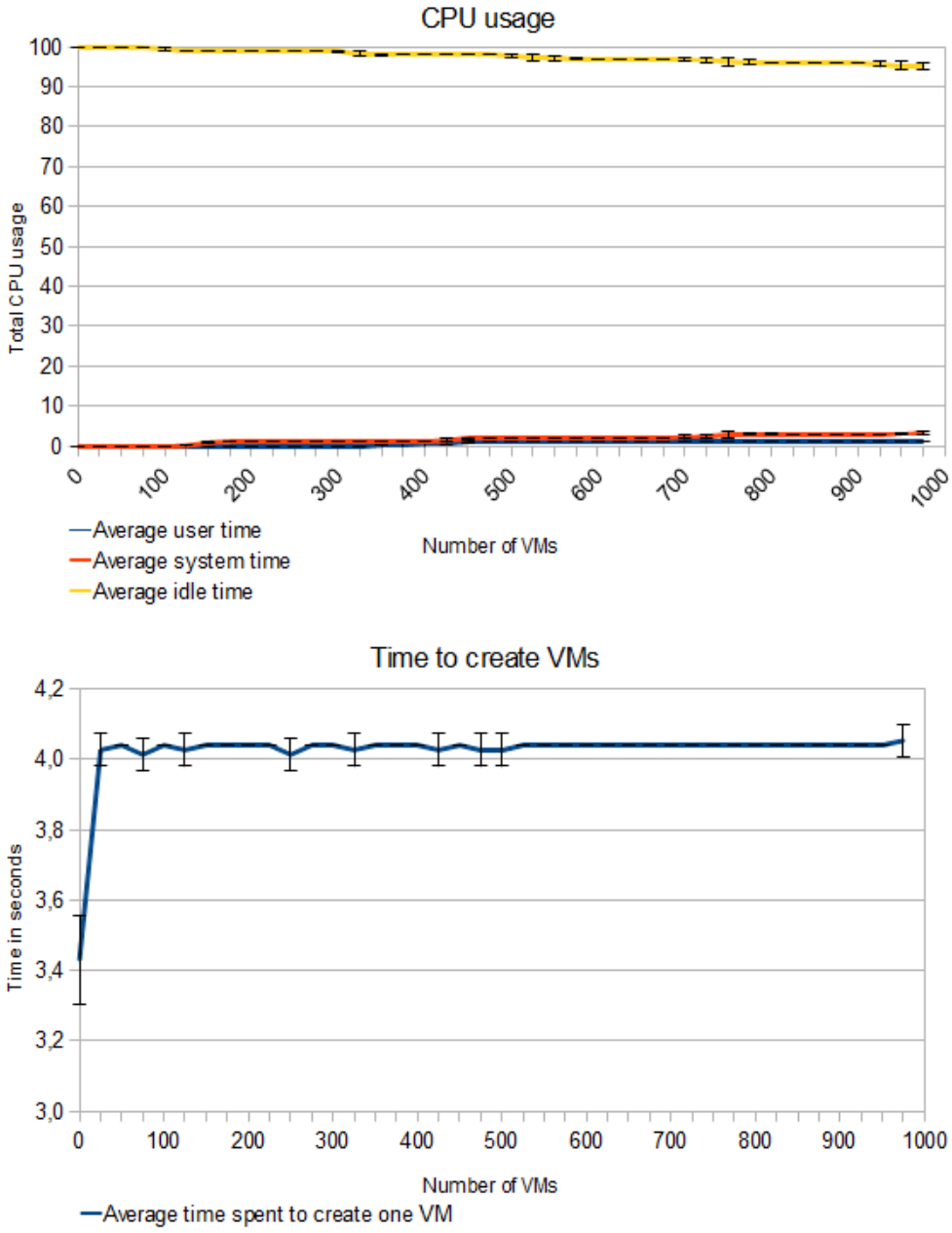

Figure 4.15: *CPU usage and creation time on the high-performance server with KVM enabled and virtual machines created through QEMU commands*

Using QEMU commands directly in order to create the virtual machines without including libvirt, results in very good performance, as seen in figure 4.15. The CPU spends most of its time idle. While the time to create new virtual machines initially is a bit longer than when using libvirt, it does not increase as the number of machines increases.

Summary

As the graphs show, the CPU usage for the same tasks on the high-performance server was much higher than on the desktop machine. Creating virtual machines through libvirt with KVM would at some point cause almost all of the CPU time to be spent on system time, while the time to create new virtual machines would increase to such a long time that the tests had to be cancelled due to time restraints. Manually designating the CPU core for each virtual machine would improve the performance somewhat. However, limiting the CPU cores used restricted the number of virtual machines which could be reliably booted.

Disabling KVM did not help the performance when using all CPU cores, but it did improve the performance as the number of CPU cores used was lowered. With KVM disabled and only one CPU core being used the system actually performs quite well. But while the host machine may perform well under these conditions, the virtual machines would likely not be able to perform well due to sharing the processing power with a large number of other virtual machines. This scenario provides very poor utilization of the available hardware resources.

4.2 Contact with the development team

In the mail correspondence with the developers, as seen in appendix E and F, they suggested that a NUMA architecture could be the cause of the issues. They believed that these issues occurred while using libvirt because of the way libvirt uses cgroups.

They also had an explanation as to what prevented creation of more than 1004 virtual machines simultaneously. Adding the following to the config file */etc/libvirt/qemu.conf* would solve the issue:

```
max_processes = <number>
max_files = <number>
```

max_processes is already a line in the file which is commented out. *max_files* needs to be added. It should also be noted that changing this number to a value higher than the ulimit for the libvirt daemon will cause libvirt not to function at all.

Overall the developers proved to be very quick to respond, and were utmost helpful.

4.3 Hypothesis testing in a reversed experiment

The test which was performed with virtual machines created through QEMU commands with manually created cgroups gave some interesting results. As shown in figure 4.15, the virtual machines created through QEMU commands

without using libvirt performed well on the high-performance server. However, with the introduction of individual cgroups for each virtual machine the performance is drastically degraded, as seen in figure 4.16

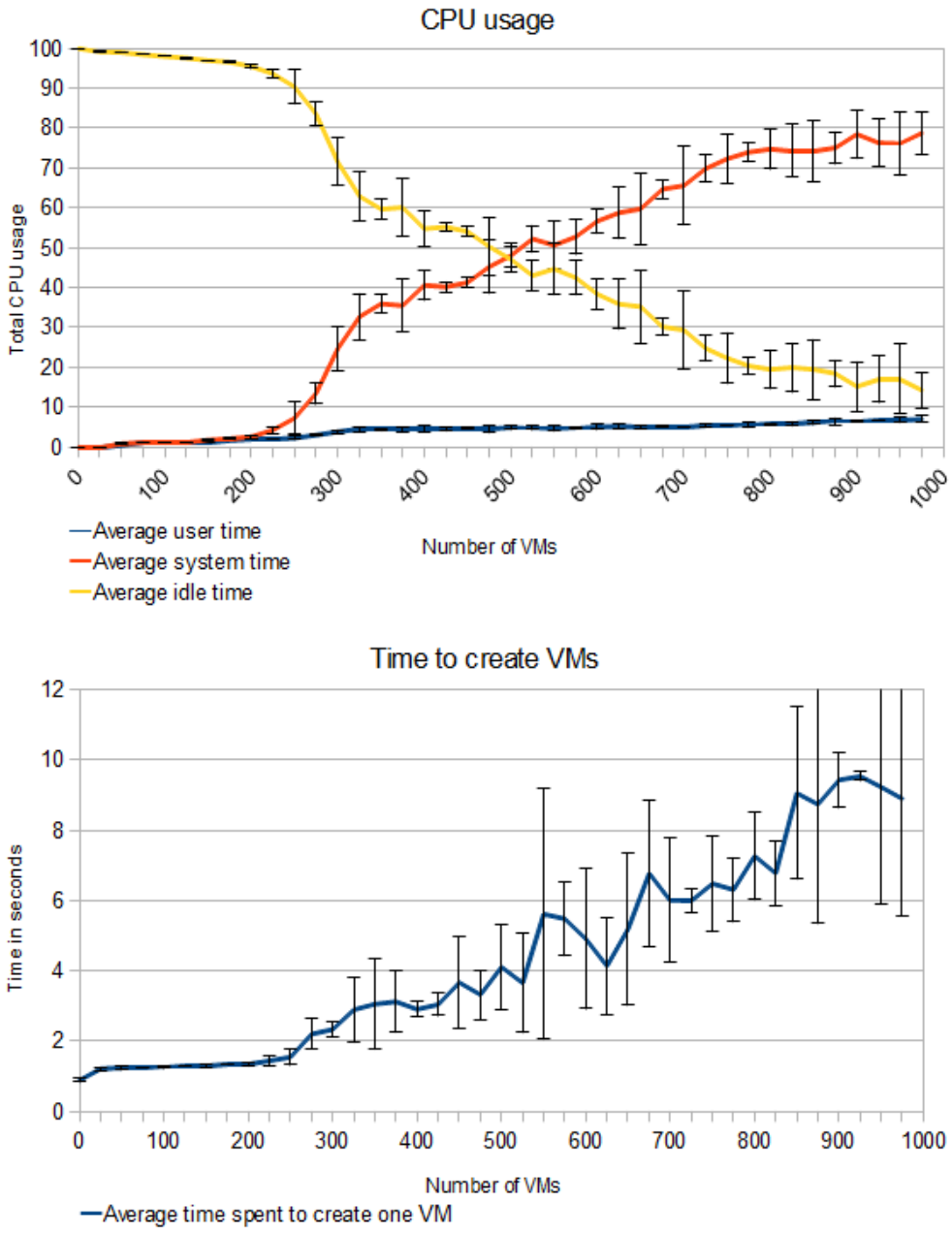

Figure 4.16: *CPU usage and creation time on the high-performance server with virtual machines created using QEMU commands with manually created cgroups*

4.4 Profiling

Profiling was performed with both Devel::NYTProf and with OProfile.

4.4.1 Devel::NYTProf

Unfortunately the profiling through Devel::NYTProf did not produce valuable results. As figure 4.17 shows, the results tells us that the majority of time within the perl script. Most of the time was spent on the line within the Sys::Virt module which creates new virtual machines. This is to be expected, but the profiling needs to go deeper than this in order to be valuable. What happens within the libvirt code or within the kernel needs to be revealed, so that it is possible to pinpoint the performance issue.

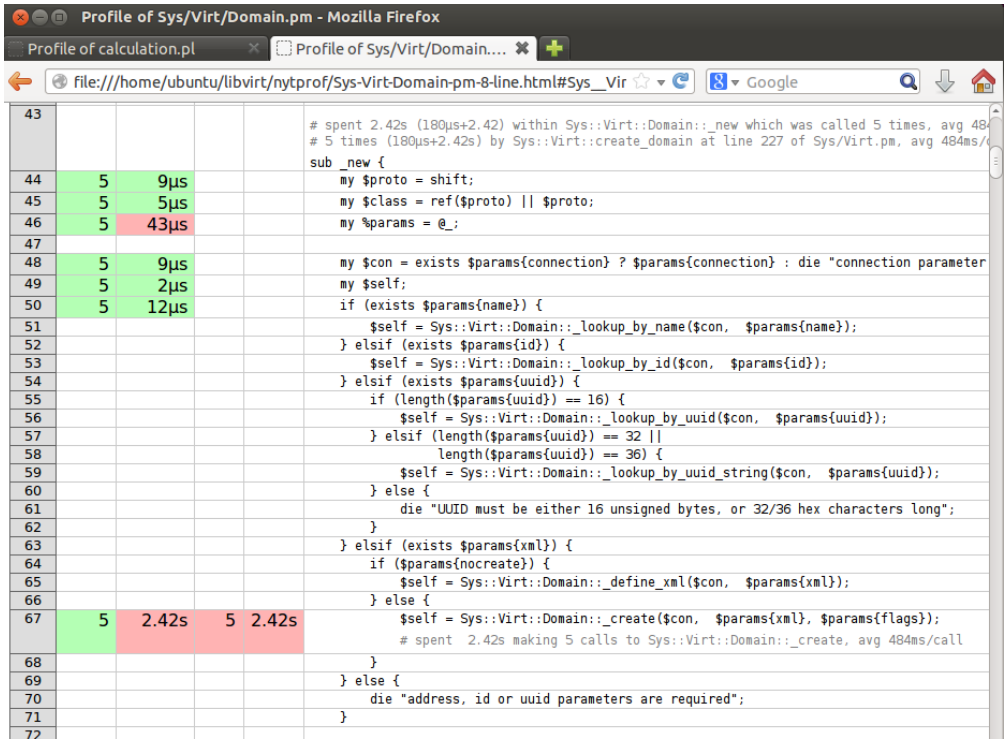

Figure 4.17: *Results from NYTProf profiling shows that much time was spent on the line which creates new virtual machines*

4.4.2 OProfile

Profiling with OProfile proved quite successful. By using the debug symbols for the kernel it was possible to see what used the majority of the CPU time. A snippet of the output from the profiling is shown below:

```
CPU: AMD64 family15h, speed 2399.89 MHz (estimated)
Counted CPU_CLK_UNHALTED events (CPU Clocks not Halted) with a
    unit mask of 0x00 (No unit mask) count 100000
```

```
samples  %        image name               symbol name
799      61.2261  vmlinux-3.5.0-17-generic __ticket_spin_lock
101       7.7395  ld-2.15.so               /lib/x86_64-linux-gnu
    /ld-2.15.so
85        6.5134  vmlinux-3.5.0-17-generic try_to_wake_up
82        6.2835  libc-2.15.so             /lib/x86_64-linux-gnu
    /libc-2.15.so
24        1.8391  vmlinux-3.5.0-17-generic tg_load_down
16        1.2261  vmlinux-3.5.0-17-generic page_fault
8         0.6130  vmlinux-3.5.0-17-generic find_vma
7         0.5364  vmlinux-3.5.0-17-generic find_get_page
7         0.5364  vmlinux-3.5.0-17-generic memset
6         0.4598  vmlinux-3.5.0-17-generic sha_transform
```

The data from several profiling runs performed at every 100th virtual machine created was used to create a graph which shows what uses the CPU time as more virtual machines are created.

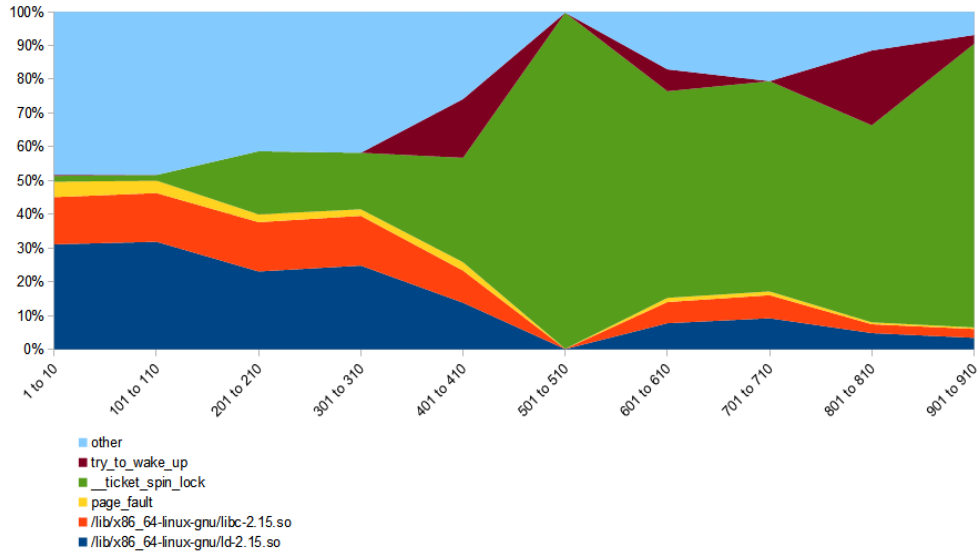

Figure 4.18: *Graph showing which processes using the CPU time when creating virtual machines. Debug symbols for the kernel makes it possible to see what is going on within the linux kernel*

The graph in figure 4.18 shows that as the number of virtual machines increases the time spent with *__ticket_spin_lock* increases by a lot, while less time is available for other tasks. At one point *__ticket_spin_lock* consumes almost all of the CPUs time.

4.5 Updated software

Experiments and profiling with OProfile was also performed on an updated version of libvirt with an updated linux kernel on the high-performance server.

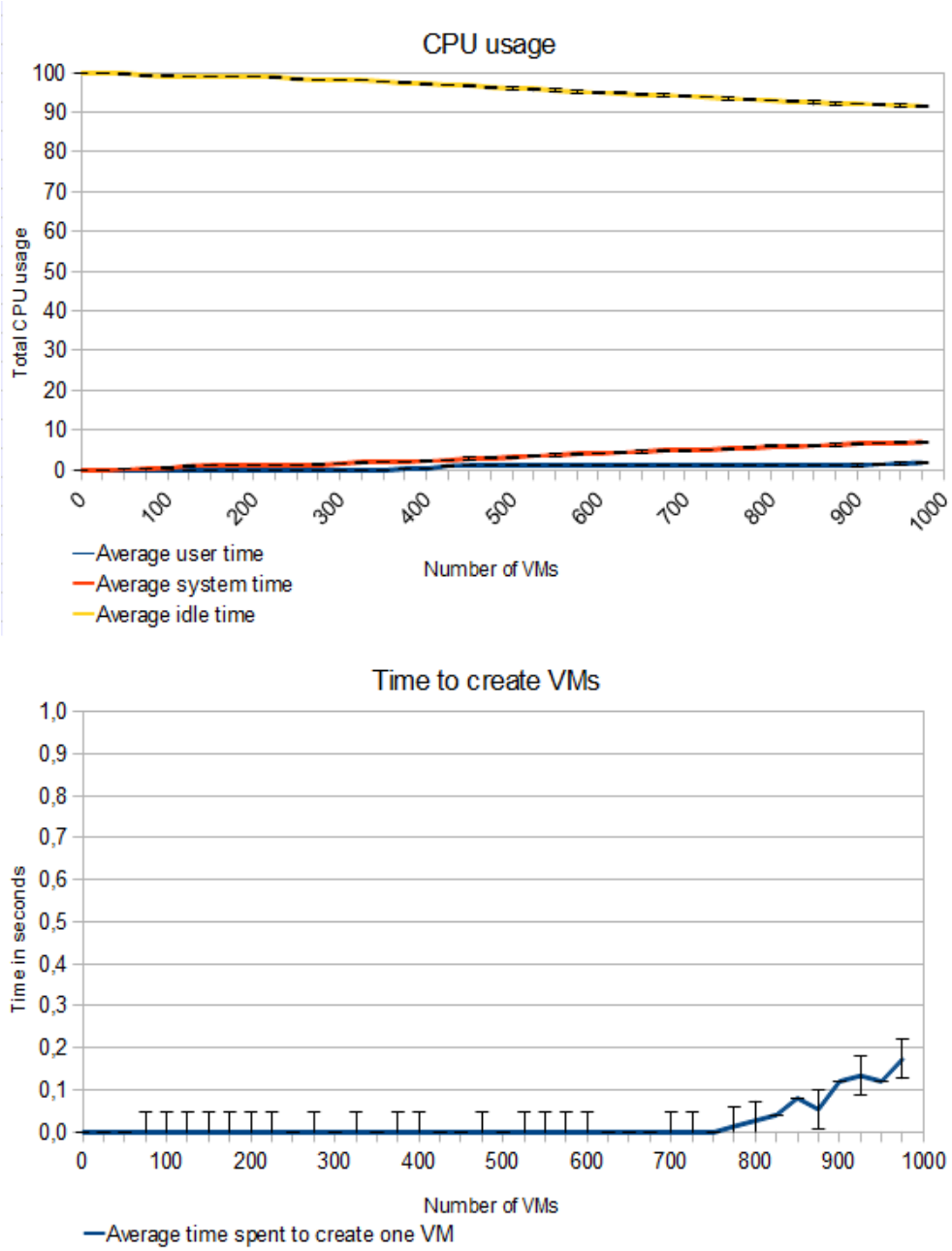

Figure 4.19: *CPU usage and creation time on the high-performance server with updated libvirt- and kernel-version*

As seen in figure 4.19, the results from the experiments with updated software are much better. The CPU idle time never goes below 90%. The time used to create new virtual machines does increase slightly towards the end, but they are still created in less than 0.2 seconds.

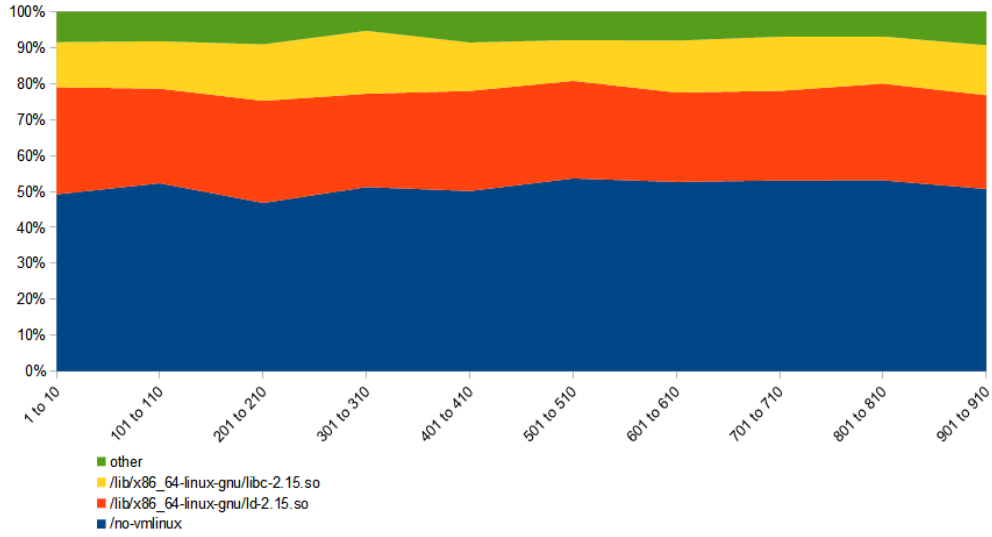

Figure 4.20: *Profiling results on the high-performance server with updated libvirt- and kernel-version*

With the updated kernel no debug symbols was found. When not using debug symbols OProfile will mark everything which happens within the kernel as *no-vmlinux*. As seen from the graph in figure 4.20, the time spent within the kernel does not increase drastically. This is different from the results seen when profiling with an older kernel version and an older libvirt version. Because the CPU time spent within the kernel does not increase drastically, more time is available for the other processes and the libvirt performance is greatly increased.

Chapter 5

Discussion and Analysis

When starting this project it was known that there were some performance issues related to creating large amounts of virtual machines through libvirt. When using libvirt to create QEMU virtual machines a high performance server would halt at around 700 machines. If the virtual machines were created through QEMU commands, a much larger number could be started. As libvirt is a commonly used management toolkit, which is essential within several software solutions, this is an important issue to resolve. This is also highlighted by the fact that the developers showed interest in the subject, as seen in appendix D.

5.1 Experiments

In order to find out more about these issues a number of experiments were started. Initially a simple desktop computer was used for the experiments. Surprisingly it performed quite well. Further on, experiments were performed on the high-performance server. The experiments confirmed what preliminary tests from before starting the project had shown, there were performance issues when using libvirt as the number of virtual machines increased. When the virtual machines was created through QEMU commands, without the inclusion of libvirt, a much larger number of virtual machines could be created without problems.

The fact that the desktop machine performed much better than a high-performance server when using libvirt came as a big surprise. This was not known when starting the project. One of the differences between the desktop machine and the high-performance server was the amount of CPU cores. In order to figure out if the high amount of CPU cores could be the cause of the performance issues experiments were performed where a limited amount of CPU cores were used.

It was also suspected that enabling KVM could cause more switches between user and kernel mode for the CPU, which could negatively affect the performance. As such, experiments were performed with and without KVM.

While the the results from the experiments showed that different settings could affect the performance on the host machine, the high-performance server

overall performed poorly when using libvirt. Manually configuring the CPU core each virtual machine should use improved performance slightly, but the desktop machine would still perform better.

With KVM disabled and only one CPU core in use the results look good, but that is not settings one would want to use to get the most out of such a powerful host machine. This graph also show that the time to create new virtual machines is gradually increasing, indicating that the system is starting to struggle.

5.2 Contact with development team

When contacting the libvirt developers through their mailing list, informing them about the results, it turned out they were aware of the issue. They suggested that a NUMA architecture could be the cause of the issue. If a virtual machine is allocated on memory split across several NUMA nodes or on a NUMA node which is not directly connected to the CPU it could be disastrous for the performance.

However, this does not explain why virtual machiness created through QEMU commands should perform much better than those created through libvirt. When asked about this, their answer was that libvirt's way of putting every VM in a cgroup was the cause of this.

In retrospect, the developers could have been contacted earlier in the project. Their input proved very valuable, and really helped the project moving forward.

5.3 Hypothesis testing in a reversed experiment

In order to check if cgroups was the cause of this a new experiment was performed. Virtual machines was created with QEMU commands, and then put into cgroups by using the *cgcreate* and *cgclassify* commands. The results from this experiment was drastically degraded performance when compared to not using cgroups. As the only new variable introduced was cgroups, this experiment strongly suggest that the problem is related to the way libvirt uses cgroups.

The results from using libvirt with KVM were still worse than the results from the test with manually created cgroups. This may indicate that there are more issues than cgroups when using libvirt on a machine with a NUMA architecture. Another theory is that the script does not accurately resemble the way libvirt uses cgroups. If this is the case, then the performance issue may not really lie within the libvirt toolkit, but rather with linux cgroups.

5.4 Profiling

Profiling was done with both Devel::NYTProf and OProfile. While the Devel::NYTProf results did not prove to be helpful, the results from OProfile

showed that *__ticket_spin_lock* consumed a large amount of CPU time.

5.4.1 Spinlocks

A spinlock allows a process to constantly check if a memory lock is available, and request a lock as soon as it becomes available. On a SMP architecture this is ”fair”, as every process will have an equal chance of getting the lock. However, on a NUMA architecture the process on CPU connected to the NUMA node holding the memory locked will have an advantage, as it has a lower latency to the memory. The feature is no longer ”fair”, as some processes will have an advantage. Under high load the processes on a CPU not directly connected to the NUMA node may experience lock starvation and poor performance.

Ticket spin lock is a feature that adds a reservation queue mechanism. This will cause processes to get the lock in the order they request it. While there is slightly more overhead for this feature, it the prevents some of the perfomance issues which may occur on a NUMA architecture with a regular spinlock. [22]

Ticket spin lock should be fair and efficient within the NUMA architecture, however the profiling shows that it takes a large amount of time. The fact that the spinlock is consuming so much time indicates that multiple processes wants to access the same memory address. Serial accessing of the memory will never be able to scale well and needs to be improved upon.

5.5 Updated software

Experiments and profiling was also performed on an updated version of libvirt with an updated linux kernel. This time the results were much better. Unfortunately a debug kernel was not found for this version, so what exactly is using the CPU-time within the kernel is not known. But the time spent on kernel features does not increase notably as more virtual machines are created. This indicates that the way the kernel handles a NUMA architecture has been improved.

When looking over prominent features introduced in kernel versions from version 3.2 to version 3.9, version 3.8 has an interesting point. A new NUMA foundation was introduced which allows for smarter NUMA policies. [26] This may be the change which has allowed for better performance.

5.6 Future Work

While the performance on the high-performance server was greatly improved on newer linux kernels, it could still be improved further. The performance when creating the virtual machines through QEMU commands is still slightly better than when using libvirt.

Additionally, when comparing the results from the desktop computer with the results from the high-performance server with updated software, the difference is still not as big as one could expect.

Chapter 6

Conclusion

The goal of this project was to find the causes for performance issues related to hosting large amounts of virtual machines through libvirt, and finding ways to mitigate these issues.

Through numerous tests the issue was found to only be relevant for massively multicore servers. Multiple settings were tested, and while they could improve performance slightly, the performance was still worse than to be expected from such powerful systems.

The results from the experiments show that there is room for large improvements on machines with a NUMA architecture. When not using libvirt, a high-performance server is able to create huge numbers of virtual machines. When using libvirt to manage the virtual machines the performance drops drastically.

Through contact with the development team, cgroups was suggested as the cause of the issue. Because of this an experiment was performed where libvirt's cgroup usage was imitated with virtual machines created through QEMU commands. This caused the virtual machines created through QEMU commands to use a lot of processing power, which they would not normally do. This indicates that cgroups is at least a part of the problem. Perhaps it would be smart if libvirt's cgroup functionality was disabled by default.

Profiling showed that *__ticket_spin_lock* consumed a lot of CPU time. This indicates serial accessing of the memory, which can not scale well with a large number of CPU cores.

Tests and profiling showed that updated software greatly improved the performance. The performance improvements look very promising, but the way software handles a NUMA architecture can still be improved upon.

Appendices

Appendix A

Perl script reporting system information while creating virtual machines

```
#! /usr/bin/perl

use strict;
use Sys::Virt;
use Data::UUID;
use Getopt::Std;

my $opt_string = 'hs:n:f:';
my %opt;
my $start;
my $end;
my $file;
my $ug = new Data::UUID;
my $idleValue = 100;
my $serial_out;

my $VM_OUTPUT = "/home/jon/libvirt/logs/logfile";
my $VM_HDA = "/home/jon/libvirt/serial_print.hda";

getopts("$opt_string", \%opt ) or usage();

usage() if $opt{h};
$start = $opt{'s'};
$end = $opt{'n'};
$file = $opt{'f'};

my $uri = "qemu:///system"; my $vmm; eval {
$vmm = Sys::Virt->new(uri => $uri); }; if ($@) {
        print "Unable to open connection to $uri" . $@->message . "\n";
}

for (my $i=$start; $i<=$end; $i++){
my $uuid = $ug->to_string($ug->create()); # Create UUID


my $xml = "
<domain type='kvm' id='1'>
  <name>microMachine-$i</name>
  <uuid>$uuid</uuid>
  <memory>16384</memory>
  <currentMemory>1</currentMemory>
  <vcpu>1</vcpu>
  <os>
    <type arch='i686' machine='pc-1.0'>hvm</type>
    <boot dev='hd'/>
  </os>
  <features>
    <acpi/>
    <apic/>
    <pae/>
  </features>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>restart</on_crash>
  <devices>
    <emulator>/usr/bin/kvm</emulator>
```

```
    <disk type='file' device='disk'>
      <driver name='qemu' type='raw'/>
      <source file='$VM_HDA'/>
      <target dev='hda' bus='ide'/>
      <address type='drive' controller='0' bus='0' unit='0'/>
    </disk>
    <controller type='ide' index='0'>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x1'/>
    </controller>
    <serial type='file'>
      <source path='$VM_OUTPUT$i.log'/>
      <target port='1'/>
    </serial>
  </devices>
</domain> ";

eval {
my $dom = $vmm->create_domain($xml); # Create VM based on XML data
};

my $check ="";
        while ($check !~ /!/){
                sleep(1);                                       # Sleep for a while after creating $
                open FILE, "$VM_OUTPUT$i.log";                  # Open the file written by VM's ser$
                $check = <FILE>;                                # Store the first line written by VM
                close FILE;                                     # Close file
        }

# Check system status
system("vmstat 3 2 > vmstat.tmp"); # Run vmstat over 10 seconds, while giving two sets of samples and write to file

sleep(1); # Short sleep before we open the file

open(VMSTAT, "vmstat.tmp"); # Open the newly written file
<VMSTAT>; <VMSTAT>; <VMSTAT>; # Skip some header lines, and the first line showing data from boot
my $vmstat = <VMSTAT>; # Store values
chomp($vmstat); # Format the vmstat data
$vmstat =~ s/ +/ /g;
$vmstat = trim($vmstat);
my @values = split(/ +/,$vmstat); # Split values into an array
close (VMSTAT);

# Retrieve time
my $time = time;

# Print results to file
open (UUID, ">>$file"); # Open uuid.lst and append new data
print "$i $time $uuid $check $vmstat\n"; # Print to screen
print UUID "$i $time $uuid $check $vmstat\n"; # Print to file
if ($@) { # Print extra if error occurs
print "Error while creating domain:" . $@->message . "\n";
print UUID "Error while creating domain:" . $@->message . "\n";
}
close (UUID); # Close uuid.lst
}

sub trim($)
{
my $string = shift;
$string =~ s/^\s+//;
$string =~ s/\s+$//;
return $string;
}

sub usage(){
        print "helpmethod\n";
        exit 0;
}

exit 0;
```

Appendix B

Perl script manually setting up cgroups for virtual machines

```
#!/usr/local/bin/perl

use strict;

my $output = "trident_cgroupstest.csv";
my $VM_OUTPUT = "/home/jon/libvirt/logs/logfile";

system("cgreate -g blkio,cpu,cpuacct,devices,freezer,memory,perf_event:/qemugroup");

for(my $i=1; $i<1005; $i++){
system("nohup qemu-system-x86_64 -m 16 -hda serial_print.hda -name qemuVM-$i -no-kvm -nographic > $VM_OUTPUT$i.log &");
#        system("nohup qemu-system-x86_64 -m 16 -hda serial_print.hda -nographic > $VM_OUTPUT$i.log &");


system("cgcreate -g blkio,cpu,cpuacct,devices,freezer,memory,perf_event:/qemugroup/qemuVM-$i");

#Find PID
sleep(1);
system("ps aux | grep qemuVM-$i > PID.file");
sleep(1);

my $file = 'PID.file'; # Name the file
open(INFO, $file); # Open the file
my @lines = <INFO>; # Read it into an array
close(INFO); # Close the file

my @firstline = split(/\s+/,$lines[0]);
print $firstline[1];

my $pid = $firstline[1];

system("cgclassify -g blkio,cpu,cpuacct,devices,freezer,memory,perf_event:/qemugroup/qemuVM-$i $pid");
print " - cgroups done\n";

        my $check ="";
        while ($check !~ /!/){
                sleep(1);                                       # Sleep for a while after creating the VM
                open FILE, "$VM_OUTPUT$i.log";                  # Open the file written by VM's serial output
        <FILE>;<FILE>;<FILE>;
                $check = <FILE>;                                # Store the first line written by VM
                close FILE;                                     # Close file
        }
print "check done\n";
        # Check system status
        system("vmstat 3 2 > vmstat.tmp");              # Run vmstat over 10 seconds, while giving two sets of samples and write to file

        sleep(1);                                       # Short sleep before we open the file

        open(VMSTAT, "vmstat.tmp");                     # Open the newly written file
        <VMSTAT>; <VMSTAT>; <VMSTAT>;                   # Skip some header lines, and the first line showing data from boot
        my $vmstat = <VMSTAT>;                          # Store values
        chomp($vmstat);                                 # Format the vmstat data
        $vmstat =~ s/ +/ /g;
        $vmstat = trim($vmstat);
        my @values = split(/ +/,$vmstat);               # Split values into an array
        close (VMSTAT);

        # Retrieve time
        my $time = time;
```

```
        # Print results to file
        open (UUID, ">>$output");                  # Open uuid.lst and append new data
        print "$i $time $check $vmstat\n";      # Print to screen
        print UUID "$i $time $check $vmstat\n"; # Print to file
        if ($@) {                                       # Print extra if error occurs
                print "Error while creating domain:" . $@->message . "\n";
                print UUID "Error while creating domain:" . $@->message . "\n";
        }
        close (UUID);                                   # Close uuid.lst

}
sub trim($)
{
        my $string = shift;
        $string =~ s/^\s+//;
        $string =~ s/\s+$//;
        return $string;
}

exit 0;
```

Appendix C

C code creating virtual machines

```
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>

int main(int argc, char *argv[])
{
    int start, stop;
printf("Number for first machine to be created:");
scanf("%d", &start);
printf("Number for last machine to be created:");
scanf("%d", &stop);

virConnectPtr conn;

conn = virConnectOpen("qemu:///system");
if (conn == NULL) {
fprintf(stderr, "Failed to open connection to qemu:///system\n");
return 1;
}

int i;

for (i=start;i<=stop;i++) {
char xml[1024];
sprintf(xml,
"<domain type='qemu'> \
 <name>microMachine-%d</name> \
 <memory>16384</memory> \
 <vcpu>1</vcpu> \
 <os> \
  <type arch='i686' machine='pc-1.0'>hvm</type> \
  <boot dev='hd'/> \
 </os> \
 <devices> \
  <emulator>/usr/bin/kvm</emulator> \
  <disk type='file' device='disk'> \
   <driver name='qemu' type='raw'/> \
   <source file='/home/jon/serial_print.hda'/> \
   <target dev='hda' bus='ide'/> \
   <address type='drive' controller='0' bus='0' unit='0'/> \
  </disk> \
  <serial type='file'> \
   <source path='/home/jon/logs/output%d.log'/> \
   <target port='1'/> \
  </serial> \
 </devices> \
</domain>", i, i);

virDomainCreateXML(conn,xml,VIR_DOMAIN_NONE);
}
virConnectClose(conn);
return 0;
}
```

Appendix D

E-mail from Eric Blake

```
> For a research project we are trying to boot a very large amount
> of tiny,custom built VM's on KVM/ubuntu. The maximum VM-count
> achieved was 1000, but with substantial slowness, and eventually
> kernel failure, while the cpu/memory loads were nowhere near
> limits. Where is the likely bottleneck?
> Any solutions, workarounds, hacks or dirty tricks?
```

Are you using cgroups? There have been some known bottlenecks in the kernel cgroup code, where it scales very miserably; and since libvirt uses a different cgroup per VM by default when cgroups are enabled, that might explain part of the problem.

Other than that, if you can profile the slowdowns, I'm sure people would be interested in the results.

Appendix E

First e-mail from Daniel P. Berrange

```
> Hi
>
> For a school project I am researching performance issues when
> using libvirt to run very large numbers of virtual machines.
>
> My experiments indicate that having more CPU cores negatively
> impacts the performance. A simple desktop with an Intel Core 2
> Duo E6550 performs better than a server with a 48 core AMD CPU.
> Limiting the cores used through 'cputune' improves performance
> on the server. If anyone has got any explanations for this,
> it would be greatly appreciated.
```

When you get machines with large numbers of CPUs, you've undoubtably also got a NUMA topology involved. To get maximum performance out of such machines you need to ensure that each VM runs entirely within one single NUMA node. If you have a VM whose memory allocation is split across NUMA nodes, or one where memory is on a different node to its CPUs, they you will absoletely ruin performance of the machine. Latest libvirt has integration with a daemon called 'autonuma' which attempts to place individual VMs within individual NUMA nods when they are started.

```
> Further on I would like to profile the slowdowns, but I am not
> sure how to approach this problem. Any recommendations on how
> to do this?
```

oprofile is one useful tool for profiling kernel + application operation.

Appendix F

Second e-mail from Daniel P. Berrange

```
> Thanks a lot for the reply.
>
> I have been comparing creating virtual machines with libvirt
> to creating them through QEMU-commands. When creating the
> virtual machiens through QEMU-commands I have not ran into
> the same performance issues as with libvirt. Why would I not
> run into the same issues with the NUMA topology
> when using the QEMU-commands?
```

The biggest configuration difference that could have an impact on performance, is that libvirt puts virtual machines into cgroups. You don't mention what OS distro / kernel version you have, but some kernel versions have had atrocious scalability with cgroups as the number of physical CPUs + # of VMs goes up. This is mostly solved in latest upstream kernels.

```
> I have also found that when I have 1004 virtual machines running,
> trying to create more through libvirt produces the error:
> "internal error invalid use of the command API".
> Do you know any reason for this?
```

Well it is a bad error message to start with. Beyond that though, most likely cause is hitting a system ulimit, perhaps max number of files ? There are some settings in /etc/libvirt/qemu.conf that let you override default ulimits - see max_processes and max_files

Bibliography

[1] Virtualbox. https://www.virtualbox.org/. Accessed: 13.02.2013.

[2] Vmware workstation 9. http://www.vmware.com/products/workstation/index.html. Accessed: 13.02.2013.

[3] libvirt virtualization api. http://libvirt.org/. Accessed: 13.02.2013.

[4] Virtual machine manager. http://virt-manager.org/. Accessed: 13.02.2013.

[5] virsh - management user interface. http://linux.die.net/man/1/virsh. Accessed: 13.02.2013.

[6] Qemu open source processor emulator. http://wiki.qemu.org/Main_Page. Accessed: 13.02.2013.

[7] Fabrice Bellard. qemu-doc - qemu emulator user documentation. http://linux.die.net/man/1/qemu-kvm. Accessed: 13.02.2013.

[8] Kernel based virtual machine. http://www.linux-kvm.org/page/Main_Page. Accessed: 13.02.2013.

[9] The perl programming language. http://www.perl.org/. Accessed: 13.02.2013.

[10] Perl.org. Comprehensive perl archive network. http://www.cpan.org/. Accessed:13.02.2013.

[11] Daniel P. Berrange. Sys::virt - represent and manage a libvirt hypervisor connection. http://search.cpan.org/~danberr/Sys-Virt-1.0.0/lib/Sys/Virt.pm. Accessed: 13.02.2013.

[12] Alexander Golomshtok. Data::uuid - perl extension for generating globally/universally unique identifiers (guids/uuids). http://search.cpan.org/~rjbs/Data-UUID-1.218/UUID.pm. Accessed: 13.02.2013.

[13] Tim Bunce. Devel-nytprof-4.23. http://search.cpan.org/~timb/Devel-NYTProf-4.23/lib/Devel/NYTProf.pm. Accessed:22.05.2013.

[14] Unknown. Oprofile. http://oprofile.sourceforge.net/. Accessed:06.05.2013.

[15] Alexander Golomshtok. Micromachines. https://github.com/alfred-bratterud/MicroMachines. Accessed:12.03.2013.

[16] Paul Menage. Cgroups. https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt. Accessed:02.05.2013.

[17] Inc. Novell and contributors. Suse linux enterprise server system analysis and tuning guide. http://doc.opensuse.org/products/draft/SLES/SLES-tuning_sd_draft/cha.tuning.cgroups.html. Accessed:02.05.2013.

[18] Janos Pasztor. Limiting linux processes cgroups explained. http://www.janoszen.com/2013/02/06/limiting-linux-processes-cgroups-explained/. Accessed:07.05.2013.

[19] John Beckett. Numa best practices for dell poweredge 12th generation servers. http://en.community.dell.com/techcenter/extras/m/white_papers/20266946.aspx. Accessed:15.05.2013.

[20] John Beckett. Non-uniform memory architecture (numa): Dual processor amd opteron platform analysis in rightmark memory analyzer. http://ixbtlabs.com/articles2/cpu/rmma-numa.html. Accessed:15.05.2013.

[21] Unknown. Numa faq. http://lse.sourceforge.net/numa/. Accessed:02.05.2013.

[22] redhat. Ticket spinlocks. https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/s-ticket-spinlocks.html. Accessed:15.05.2013.

[23] Kvm/installation. https://help.ubuntu.com/community/KVM/Installation. Accessed: 14.02.2013.

[24]

[25] Janning. Debian: Too many open files. http://serverfault.com/questions/20387/debian-too-many-open-files. Accessed:12.03.2013.

[26] Diego Calleja. Linux 3.8. http://kernelnewbies.org/Linux_3.8. Accessed:15.05.2013.